NPS ARCHIVE
2000.03
UYSAL, H.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A MODEL FOR GENERATION AND PROCESSING OF LINK STATE INFORMATION IN SAAM ARCHITECTURE

by

H.Huseyin Uysal

March 2000

Thesis Advisor: Geoffrey Xie
Co-Advisor : Bert Lundy

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2000	Master's Thesis

TITLE AND SUBTITLE : A Model for Generation and Processing of Link State Information in SAAM Architecture

5. FUNDING NUMBERS

6. AUTHOR(S)
H.Huseyin UYSAL

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA and NASA

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
G417

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis presents a model of link state advertisement generation for the SAAM (Server and Agent Based Network Management) architecture. The model includes generation and processing of link state data. In a SAAM network, a central server manages a region of 20-40 lightweight routers. The server learns the link performance of the routers from processing Link State Advertisement messages that are periodically sent by the routers. The server uses the information to maintain a Path Information Base to manage routing within the region. A router also sends a triggered Link State Advertisement message when one of its interfaces fails.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Quality of Service, Networks, Flows, Link State Advertisement	111
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	19. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 **Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18**

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A MODEL FOR GENERATION AND PROCESSING OF LINK STATE INFORMATION IN SAAM ARCHITECTURE

H. Huseyin UYSAL
1st Lt., Turkish Army
B.S., Turkish Military Academy, 1992

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

NPS Archive
2000.03
Uysal, H.

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis presents a model of link state advertisement generation for the SAAM (Server and Agent Based Network Management) architecture. The model includes generation and processing of link state data. In a SAAM network, a central server manages a region of 20-40 lightweight routers. The server learns the link performance of the routers from processing Link State Advertisement messages that are periodically sent by the routers. The server uses the information to maintain a Path Information Base to manage routing within the region. A router also sends a triggered Link State Advertisement message when one of its interfaces fails.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. OVERVIEW OF SAAM ARCHITECTURE 1
1. SAAM Region 2
2. The Path Information Base and Service Level Pipes 3
3. Hierarchical Organization of the SAAM Servers 4
B. BENEFITS OF SAAM 5
1. Integrated and Differentiated Services 5
2. Fault Tolerance 6
3. Auto-configuration of Control Channel 6
4. Scalability and Incremental Deployment 6
C. SCOPE OF THIS THESIS 6
D. MAJOR CONTRIBUTIONS OF THIS THESIS 7
E. ORGANIZATION 7
II. BACKGROUND 9
A. ROUTING 9
1. Scalability 9
2. Size of Routing Table 10
3. Robustness and Adaptability 10
4. Path Optimization And Load Balancing 10
B. ROUTING ALGORITHMS 11
1. Distance Vector Routing 12
a. Routing Information Protocol (RIP) 14
2. Link State Routing 15
a. Open shortest path first (OSPF) 16
C. ATM PNNI (PRIVATE NETWORK-NETWORK INTERFACE) 19
III. LINK STATE ADVERTISEMENT MODEL 23
A. REQUIREMENTS ANALYSIS 24
1. Goals 24
2. QoS Parameters 25
B. QOS PARAMETERS OF SERVICE LEVEL PIPE 25
1. QoS Parameter Measurements 26
a. Utilization 26
b. Delay 28
c. Loss Rate 29
2. Data Collection at SLP 29
3. Service Level Pipe State Advertisement 31
C. INTERFACE STATE ADVERTISEMENT 32
D. LINK STATE ADVERTISEMENT 34
E. INTERFACE FAILURE 35
IV. IMPLEMENTATION 37
A. LSA GENERATION 37
1. PriorityQueue 37
2. Service Level Pipe State Advertisement (SLP-SA) 40
3. Interface State Advertisement (InterfaceSA) 40
4. LinkStateMonitor 42
5. LinkStateAdvertisement 42
6. LsaGenerator 43
B. LSA PROCESSING 46
1. ServerAgent 46

2. Server 46
a. void processLSA(LinkStateAdvertisement lsa) 46
b. void checkAndAdd(int nodeId, InterfaceSA curLsa) 48
c. void updatePIB(int nodeId, InterfaceSA curLsa) 48
d. void removeInterfaceFromPIB(int nodeId,InterfaceLSA curLsa) 48
e. void removePathsTraversingInterface(IPv6Address ip) 48
f. void removeLinkFromPIB(IPv6Address ip) 49
g. void removeInterfaceFromNode(IPv6Address ip) 49
h. void checkRouterId(IPv6Address routerId,Vector iLsaVector) 49
C. SIMULATION OF INTERFACE FAILURE 49
V. INTEGRATION AND TESTS 53
A. INTEGRATION 53
B. TESTS 55
VI. CONCLUSIONS 57
A. SUMMARY 57
B. LESSONS LEARNED 57
C. FUTURE WORK 58
1. Improving the PIB Path Processing 58
2. Rerouting of the Flows If an Interface Fails 58
3. Securing SAAM 58
APPENDIX A. PRIORITYQUEUE CLASS SOURCE CODE 59
APPENDIX B. LINKSTATE MONITOR CLASS SOURCE CODE 67
APPENDIX C. LSAGENERATO CLASS SOURCE CODE 69
APPENDIX D. LINKSTATEADVERTISEMENT CLASS SOURCE CODE 77
APPENDIX E. INTERFACELSA CLASS SOURCE CODE 81
APPENDIX F. SLPLSA CLASS SOURCE CODE 85
APPENDIX G. INTERFACEFAILURE CLASS SOURCE CODE 89
APPENDIX H. SOURCE CODE OF LSA PROCESSING METHODS ADDED TO THE SERVER 91
LIST OF REFERENCES 97
INITIAL DITRIBUTION LIST 99

ACKNOWLEDGEMENTS

The author would like to acknowledge Prof. Geoffrey XIE for his continuous support throughout all the phases of this thesis. Without his help I could not finish this thesis. He always encouraged me in my study, even debugged the code with me.

I am also grateful to my wife, who supported me by her love and patience throughout my study.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Today's Internet only provides Best Effort service, where there is no guarantee of timely delivery. With the growth of Internet, demand for real-time traffic support is increasing. There is a need for a network management system that supports different classes of quality of service (QoS).

Server and Agent based Active Management (SAAM) is a proposal developed to realize the goals of the Next Generation Internet (NGI) initiative, which will support all service classes required by the future.

A. OVERVIEW OF SAAM ARCHITECTURE

The current network architecture depends on standalone routers for network management tasks. Every router tries to know the network topology and determine every possible path to every other router in the network. This workload on routers must be minimized to provide quality of service for real-time applications.

SAAM is designed to provide QoS (Quality of Service) support using today's widespread IP networks. SAAM uses an approach that takes most of the network management tasks from the routers and gives them to a dedicated server, called SAAM server. With this architecture real time network services will be provided in an efficient manner, while dealing with the dynamic changes in the network environment. SAAM will enable networks to ultimately provide integrated services.

1. SAAM Region

A SAAM region typically consists of 20-40 lightweight routers and a server. Server performs most of the network management tasks on behalf of the routers.

To perform network management tasks effectively server collects information from routers in the region and maintains a Path Information Base (PIB) [Figure 1.1]. The server computes and identifies all valid paths between edge routers, that physically connect customer networks or other SAAM regions, and stores them in the PIB. When a flow request is received, the server uses the PIB to try to find the best path that supports the QoS metrics of the request. The server then informs the requesting host of the result with a flow response message. If the flow can be supported, the server updates the states of the routers on the flow path adding the necessary routing entry for the flow to their routing table, with a flow routing table update message.

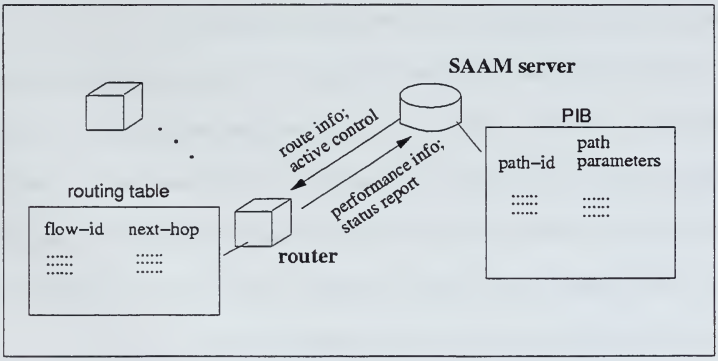


Figure 1. 1. SAAM Region and Building the PIB (Path Information Base) [From Ref.6].

2. The Path Information Base and Service Level Pipes

The server maintains the network topology and routing information in its PIB. The PIB includes every valid path from each source to each destination in the region. Every path has specific QoS parameters associated with it. In an integrated services network requires each link shared by a set of logical service level pipes (see Figure 1.2). Each of these service level pipes provides a particular level of network performance primarily measured by packet delay, and packet loss rate. [Ref. 7]

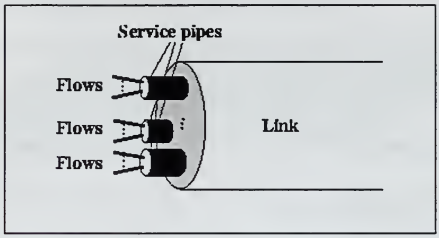


Figure 1. 2. Link and Service Level Pipes [From Ref.6]

Each flow request is made for certain type and level of network QoS. The source will invoke a reservation protocol such as RSVP to establish a flow to the destination. This request is translated into SAAM format by the edge router used by the source. The edge router then forwards to the server. The server consults the PIB and tries to find a series of service level pipes whose composite QoS meets the requirements of the flow. Since routes are calculated at the server, there is no need for routers to handle change in the network conditions. Their resources can be dedicated to packet forwarding and value added services (e.g. packet authentication).

QoS management for the NGI should handle the dynamically changing network conditions. The QoS requirements of real-time data are measured in milliseconds. When a degradation of the performance of a path causes violation of the QoS requirements of active flows. New paths should be assigned to the affected flows. All this demands the network management system to be proactive, able to detect and react to changing network conditions within fraction of a second. [Ref. 8]

3. Hierarchical Organization of the SAAM Servers

SAAM organizes the servers in a hierarchy to address the scalability issue. SAAM partitions the network into autonomous regions each consisting of a limited number of routers. One server is assigned to each region, managing all routers. The server needs an accurate picture of the region to manage real-time traffic. The PIB should be updated regularly to maintain accuracy. More frequent updates mean more accurate PIB. But updates consume network bandwidth. A compromise should be reached between the level of accuracy and consumed bandwidth.

In Figure 1.3, the SAAM network has a two level hierarchy. Each server controls the immediate children of its region, which can be a router or a server of a lower level region. A base level server gathers link performance information from routers and advertises itself as a router to top level server.

A base level server only responds flow requests forwarded by an edge router in its region. If a flow needs to traverse multiple regions, a higher level server will also be included in processing request. The higher level server establishes a flow through regions

and distributes the QoS requirements of the flow among the regions. The server of each makes the routing decisions inside the region.

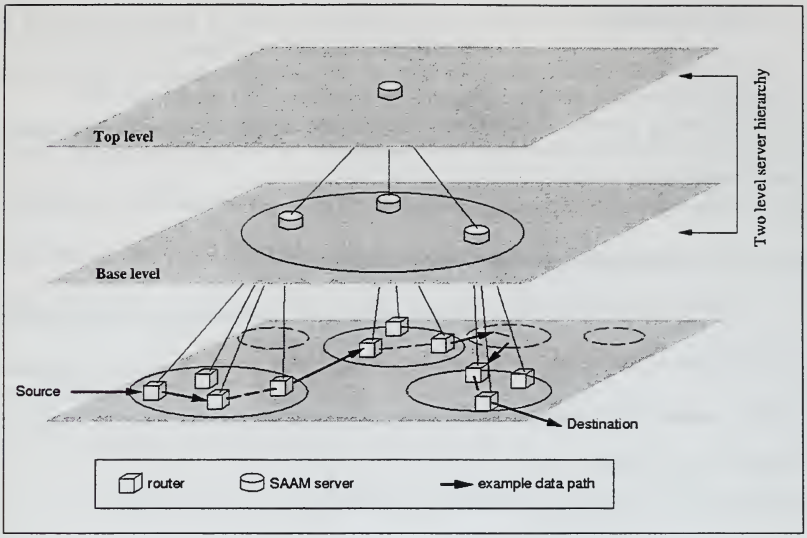


Figure 1. 3. Hierarchical Organization of the SAAM Servers [From Ref.6].

B. BENEFITS OF SAAM

The goal of SAAM is to provide QoS for real-time traffic flows using a central resource management scheme. Benefits of SAAM include:

1. Integrated and Differentiated Services

SAAM will support both Integrated and Differentiated services in addition to Best Effort service. SAAM uses a service level pipe for each service class and allocates resources according to user requirements.

2. Fault Tolerance

The server is the main player in each SAAM region. It performs admission control and allocate resources. A failure of the server will have a dramatic effect on the performance of the region. A failure must be detected timely and recovery actions be taken as soon as possible. SAAM uses a *backup server* that will manage the region in case the *primary server* fails.

3. Auto-configuration of Control Channel

Server builds up a control channel between the server and the routers, that is a spanning tree of the region in which the control traffic flows, by auto-configuration. Control channel is refreshed at small intervals, so that server can adapt to changes in network status quickly.

4. Scalability and Incremental Deployment

SAAM is designed to support hierarchical routing (see Figure1.3). Servers may be deployed incrementally, providing improvements of network performance to ISPs that use SAAM.

C. SCOPE OF THIS THESIS

The goal of this thesis is to develop an efficient scheme for generation and processing of Link State Advertisement (LSA) messages, and to integrate this scheme with the existing SAAM architecture. The SAAM server needs to have an accurate picture of the current performance of the routers to make good routing decisions. This

includes a LSA generation method that will keep the server informed about the current network performance.

D. MAJOR CONTRIBUTIONS OF THIS THESIS

This thesis adds LSA generation and processing to the SAAM architecture. A router monitors the link state of its interfaces and reports the state information by LSA messages to its SAAM server. The server processes LSA messages and updates its PIB accordingly. With periodic LSA updates, the server always has up-to-date information about the network status.

E. ORGANIZATION

This thesis is organized with the following chapters:

- Chapter II presents the terminology information of existing routing algorithms and their link state advertisement generation strategies.
- Chapter III describes our Link State Advertisement generation and processing model.
- Chapter IV describes the implementation of the model.
- Chapter V summarizes the implementation results, lessons learned, and the future work.

THIS PAGE INTENTIONALY LEFT BLANK

II. BACKGROUND

A. ROUTING

A node in a network often needs to know whether another node in the same network or in another network is reachable. The node also should know the exact direction (next hop) to use for a reachable destination. There must be a network wide mechanism in place to provide the necessary information. This mechanism is called *routing*.

Routing is accomplished by means of *routing protocols* that establish mutually consistent *routing tables* in all the nodes. Each node builds up its routing table from information about its own links and data received from other nodes. From the routing table, each node calculates *best paths* to reach other nodes or networks. Based on the calculations, each node creates a table called *forwarding table,* which contains the appropriate next hop to each reachable destination. Later when the node receives a packet, it consults the forwarding table using the packet's destination address as key, and forwards the packet to the next hop returned for that destination. [Ref. 2]

There are several issues that routing protocols must address:

1. Scalability

Requiring the whole network topology to build routing tables may not be that big a problem for a small to mid-sized network. But when the network size is big, it is practically impossible for a node to know the whole topology and make routing decisions

in an efficient manner. Routing protocols should use hierarchical routing to scale well in big networks.

2. Size of Routing Table

The size of a routing table increases as the network size increases. So does the overhead to maintain the routing table. When dealing with a big network, aggregation of topology data is necessary to reduce the memory space used by routing tables. Hierarchical routing localizes the affect of a network topology change. This in turn decreases the route calculation and lookup overhead as well as communication cost associated with topology information exchanges among nodes.

3. Robustness and Adaptability

Routing algorithms should avoid creating *loops* and *oscillations* in the network. They should protect themselves by periodic consistency tests. When a topology change occurs, routing algorithms should respond to the change and converge fast to avoid loops and oscillations.

4. Path Optimization And Load Balancing

A selected path should be optimal regarding some performance metric. A best path may not necessarily be the shortest path. Depending on the metric, it may be the path with the least delay, or with the lowest monetary cost. Routing algorithm should use more than one path for each destination and distribute the network traffic among these paths to balance the load and increase the network performance. [Ref.1]

B. ROUTING ALGORITHMS

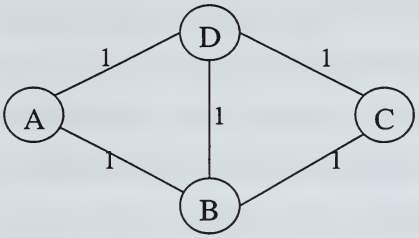
Routing algorithms can be classified into two major classes: *distance vector routing* and *link state routing* algorithms. They have the following features:

- They are distributed algorithms; every node in the network runs these algorithms independently.
- They assume that a node knows its neighboring nodes and the cost of reaching to each neighbor node.
- They both construct a routing table that contains an entry per destination. Each entry consist of the route cost and the next hop to the destination.

Each node exchanges topology information with neighbors using update packets. The circumstances to create update packets are similar in both algorithms. One of these circumstances is the *periodic update*. Each node automatically sends an update packet at each period. Periodic updates enable the neighboring nodes to know that the node is alive. Another circumstance is the *triggered update*. Although both algorithms use triggered updates, the triggering events are different. In distance vector routing, a node generates a triggered update packet whenever it receives an update that causes a change in its routing table. In link state routing a node generates triggered updates when it detects that one of its links or neighbors is down. [Ref.2]

1. Distance Vector Routing

As mentioned, every node is assumed to know its neighbor nodes at start. Each destination is initially assigned a cost of infinite, except for the neighbors. Nodes running distance vector algorithm exchange entire routing tables with neighbors. When a node receives a routing table from one of its neighbors, it uses that table to update its own. If the cost to a destination in the neighbor table plus the cost of the link to the neighbor is less than the cost given in its own table, the node updates its routing table by designating the neighbor node as the next hop to that destination.



Dest	Cost	N.H.
B	1	B
D	1	D

Dest	Cost	N.H.
B	1	B
C	2	B
D	1	D

Dest.	Cost	N.H.
B	1	B
C	2	B
D	1	D

Figure 2. 1. Distance Vector Algorithm on Node A.

In Figure 2.1, at start *A* only knows the distance to its neighbors (left table). After receiving *B*'s table it learns that it can reach *C* by a distance of 2. *A* adds *C* to its table with *B* as the next hop (middle table). *D*'s table does not change *A*'s table, since the advertised distance to reach *C* is not less than the current distance to *C* (right table).

Since table exchanges are done hop by hop, distance vector algorithms suffer from slow recovery from link failures, and during the recovery time routing loops may occur.

Figure 2. 2. Count to Infinity Problem.

Distance vector routing also suffers from a *count to infinity* problem. This problem can occur between two neighboring nodes. In Figure 2.2, node B detects that the link to C is down and updates its table. B sends its table to A, but at the same time it receives A's table showing a cost of 2 to reach to C, less than the one it has currently. So it updates the table with a cost of 3 and next hop as A, and advertises its table to A. In the previous table exchange A has learned the link failure but when it receives B's table it updates its routing table with a cost of 4 and B as the next hop. This situation continues until the cost reaches infinite. [Ref. 1]

One of the techniques for preventing *count to infinity* is named *split horizons*. In this method nodes will not advertise a route back to the same node they received the update. Another technique is called *split horizon with poison reverse*. This technique advertises the update back to the neighbor but with a negative information to ensure it won't use the update to count to infinity. Both techniques prevent count to infinity between two neighboring nodes, but they can not address a count to infinity problem among more than two nodes. [Ref. 2]

a. Routing Information Protocol (RIP)

Routing Information Protocol is the most widely used routing protocol in the Internet today. RIP is a distance vector algorithm. It uses hop count as the cost metric to calculate the best routes. So RIP tries to find the minimum-hop route to a destination.

Every link is assigned a distance (cost) of one. Valid distances are between 1 and 15, 16 representing the infinity. The metric does not take the available bandwidth or the link utilization into account. So RIP will prefer a three hop 10MBps route to a 5 hop 10Gbps route. This makes RIP suitable for small networks where simplicity of implementation and configuration is more important than performance requirements.

Nodes running RIP send periodic updates every 30 seconds by default. If a node does not hear from a neighboring node for three update cycles (180 sec.), it declares that the neighbor is down. RIP protocol uses split horizon with poison reverse to avoid the count to infinity problem. [Ref 1] Also when a node's table changes because of another node's update, it sends a triggered advertisement to its neighbors.

RIP can be configured to use different routing metrics. It can support multiple address families aside from IP. Unlike RIP version 1, RIP version 2 has some functionality that supports hierarchical routing.

2. Link State Routing

In link state routing, like distance vector routing, a node is capable of finding out the state of each of its links including the link cost. Each node disseminates this information to its neighbors. Each neighbor in turn forwards the information to its neighbors except for the one it get the information from. This way the network is flooded by the link state information of each node and every node has enough knowledge of the network to build a complete map of the network. Link state routing relies on controlled flooding of this link-state information since it calculates routes from the sum of link state knowledge.

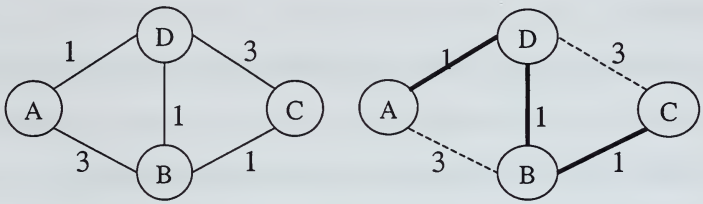
Each node prepares a packet called *Link State Advertisement* (LSA) that contains the id of the node, a list of neighbors and the cost to each neighbor, a sequence number, and a time to live. This LSA packet will be stored at each neighboring node for the *time to live* period duration. When this period expires, the LSA is removed. This ensures that an old LSA will be removed from the network eventually.

The sequence number and time to live are used to make the link state routing reliable. The sequence number ensures that each node has the latest LSA from each neighbor. A node simply discards any incoming LSA that has an older (smaller) sequence number than the LSA that is currently stored in the node.

When a node knows the whole topology, it can find a physical path to any reachable destination. The node, using the database of LSAs first builds a spanning tree where the node itself is the root. Dijkstra's shortest path algorithm is used to form the spanning tree. The algorithm uses a set, P, for the reachable nodes. Every node, that

should be in P must be reachable using a path started with a node already in P. Algorithm starts with adding the root to P and continues until every node is added to P. [Ref.1]

Figure 2.3 shows the link state algorithm on node *A*. At start *A* only knows the distances to *D* and *B*. When it receives the LSA of *B*, it learns that it can reach *C* with a cost of 4, using *B* as the next hop. Later after receiving and processing *D*'s LSA, *A* learns that it can reach to *C* with a smaller cost. It updates its routing table with *D* as the next hop for all entries. Even though *B* is a directly connected neighbor, *A* uses *D* as the next hop to *B* since the route as smaller cost.



Destination	Cost	Next Hop
D	1	D
B	2	D
C	3	D

Figure 2. 3. Link State Routing on Node A.

a. Open shortest path first (OSPF)

OSPF is a link state routing protocol of choice in today's networks. OSPF is designed to run internal to a single autonomous system. Each OSPF router maintains an identical database describing the autonomous system's topology. From this database, a routing table is calculated by constructing a shortest path tree. OSPF recalculates routes

quickly in face of topological changes, incurring a minimum of routing protocol traffic. [Ref. 1]

OSPF allows sets of subnetworks to be grouped together, adding additional hierarchy within an AS. Such a grouping is called an *area* and its topology is hidden from the rest of the autonomous system. This information hiding enables a significant reduction in routing traffic. An area is a generalization of an IP sub-net. Area border router advertises the link state advertisements that describe the area. [Ref. 1]

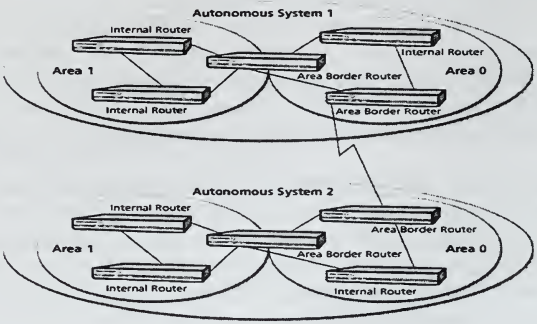


Figure 2. 4. OSPF Areas in Autonomous Systems.

All OSPF routing protocol data exchanges are authenticated. This means that only trusted routers can participate in the AS's routing. This is a nice feature since a misconfigured router can advertise that it can reach every destination in the network with a cost of 0. This could eventually bring the whole network to a halt. OSPF allows use of different authentication methods. There is an "Authentication Type" field in the LSA

header. Users can set this value according to their needs and come up with their own authentication scheme. [Ref. 2]

OSPF routers use a sub-protocol, named *Hello Protocol,* to establish and maintain neighbor relationships. Hello messages are also used to detect that a neighbor is alive. A router detects the failure of a neighbor router if the neighbor router does not acknowledge the Hello message. When a router detects a failure, it generates a triggered LSA and the network is flooded with the LSA. Each router in the network recalculates its routing table and network converges quickly.

The Hello protocol is also used to select a designated router (DR) and a back up DR for multi-access networks. Only the DR generates link state advertisements. The DR concept enables a reduction in the number of adjacencies required on a multi-access network. This in turn reduces the amount of routing protocol traffic and the size of the topological database. When a router has an LSA to propagate on the network, it transmits the LSA to multicast address of all DRs, to which both the DR and the backup DR listen. When the DR receives the LSA, the DR multicasts the LSA to all the other routers.

The OSPF routing algorithm is based on Dijkstra's shortest path algorithm. It tries to find a path that is optimal with respect to a performance metric. The metrics can be latency, hop count, monetary cost, or combination of them.

Another nice feature of the OSPF is that it allows multiple equal-cost routes to the same destination. Routers can choose different routes when forwarding

packets to the same destination. Traffic load in the network can be balanced with the use of multiple routes. [Ref.1]

C. ATM PNNI (PRIVATE NETWORK-NETWORK INTERFACE)

PNNI is a hierarchical routing protocol designed by the ATM Forum to support different hierarchies and topology aggregation. PNNI uses link state routing approach. PNNI supports many levels of hierarchical routing and allows routing based on quality of service [Ref.1]. Support of many levels is an important characteristic. In global routing a node has all the knowledge about the network. But often it is not feasible that a router maintains full information for all physical links. A node should maintain aggregated information about the network. Using this information, the node can find a sequence of partial paths at different abstraction levels. [Ref.3]

Nodes at each level form a *peer group*. A peer group is expected to have between ten and a hundred nodes [Ref.1]. The nodes in the same peer group elect a *peer group leader*, which will function as *Logical Group Node* (LGN) representing the peer group as a single logical node in the next higher hierarchical level. PGL summarizes the topology information in the peer group and distributes it in the next higher level. This in turn reduces the amount of the topology database traffic. [Ref.4]

LGNs at the same hierarchical level are connected by logical links and form another peer group. LGNs behave just like lower level nodes and synchronizes the LGN database with the neighboring LGNs. Then database is flooded in the logical peer group. PNNI in this way can support 100 different levels. [Ref.4]

PNNI has three basic protocols that define the dynamic distribution of routing information and the signaling, once the routing information is complete. They are Hello, Database Synchronization, and Peer Group Leader Selection.

A node uses the Hello protocol to discover the neighbors and to maintain the connectivity to the neighbors. By Hello protocol a node learns the peer group id, node identity, and ATM address of its neighbor nodes. Hello protocol operates on a per link basis. As explained before, another use of the Hello Protocol is to detect the failure of a neighbor node and generate a triggered update accordingly.

Once a node discovers that a neighbor node is in the same peer group it starts the process of synchronizing the topology database with each other. After the database synchronization process is complete, both nodes will have the same database. This information is then flooded through the entire peer group by *PNNI Topology State Elements* (PTSE) messages. ATM switches will have the same topology database after flooding. Each PTSE has an age. PTSE is valid during the age period. If the age of a PTSE expires, it will be removed from the topology database. A switch periodically sends update PTSEs to the others. When a new PTSE is received it will replace the old one. [Ref.4]

PNNI uses source routing for connection reservation through the network. The first switch computes the entire path according to its topology database. The first switch can make a fairly good decision abut the path to follow since QoS parameters are kept in the database along with topology information. If a switch accepts the connection it passes

the request to the next one in the path. The use of source routing eliminates routing loops and speeds convergence in case of a link or node failure. [Ref.1]

THIS PAGE INTENTIONALY LEFT BLANK

III. LINK STATE ADVERTISEMENT MODEL

Existing routing protocols are mostly concerned with topology information, disregarding the availability of network resources. Thus they all have limitations in providing guaranteed or differentiated quality of service (QoS). To provide QoS, routing protocols should reserve network resources, such as link bandwidth, for flows or service classes. This will bring much more constraints to QoS routing algorithms compared to current shortest path algorithms. [Ref.5]

Each SAAM server maintains a Path Information Base for its region and performs network management tasks based on this database. Each router periodically generates Link State Advertisement (LSA) packets and use a special control channel to send them to the server. The server processes received LSA packets to update its PIB and make necessary decisions.

SAAM has an auto-configuration feature that establishes a control traffic channel between the server and the routers. The SAAM server periodically generates a *Downward Configuration Message (DCM)* to update the control channel. A router learns the way to reach to the server after receiving the DCM message. The router in turn multicasts the DCM message to all its neighbors, except for the one from which it has received the DCM. At the last step, the routers send the reachability information of the control channel back to the server using an *Upward Configuration Message (UCM)*. Figure 3.1 demonstrates the steps in the auto-configuration of the SAAM control channel.

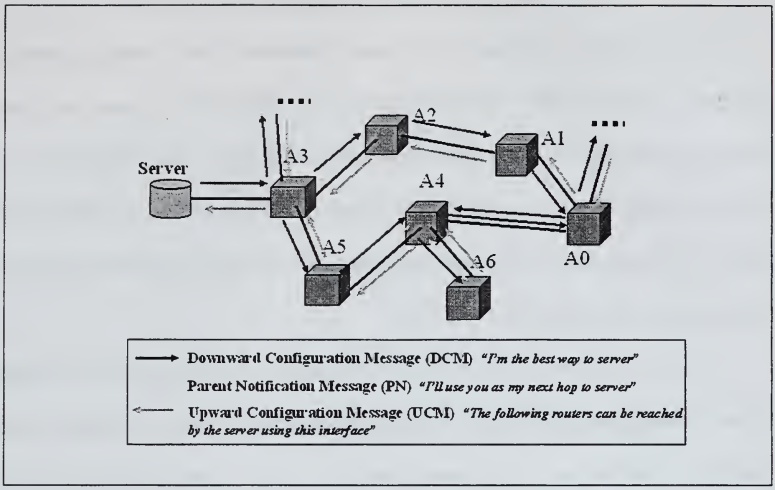

Server
A3
A2
A1
A4
A5
A6
A0
Downward Configuration Message (DCM) "I'm the best way to server"
Parent Notification Message (PN) "I'll use you as my next hop to server"
Upward Configuration Message (UCM) "The following routers can be reached by the server using this interface"

Figure 3. 1. Auto-configuration and Building the Control Channel.

A. REQUIREMENTS ANALYSIS

1. Goals

The main goal of LSA generation is to deliver timely link state data to the server. The major requirements are:

- Keep the SAAM server up-to-date about available network resources.
- Minimize the bandwidth requirements and processing overhead of Link State Advertisement packets.
- Detect local interface failures and inform the server of such failures.

- Generate an extra LSA in case of a dramatic change in the performance of a particular type of traffic. For example, an LSA should be sent to the server when the queues for real-time flows build up rapidly because of a bug in the packet schedular.

2. QoS Parameters

The server maintains the current state of the network in a Path Information Base (PIB). The PIB contains data describing QoS capability of each service class as well. Specifically these QoS parameters are:

- Utilization: The percentage of time that a service class is busy during a predefined time period.
- Delay: The average time that a packet waits in queue before transmission.
- Loss Rate: The percentage of packets that are dropped because of buffer overflow.

A router must measure these parameters at each of its interfaces. The measurements from different interfaces are combined to form an LSA packet.

B. QOS PARAMETERS OF SERVICE LEVEL PIPE

Each service level pipe carries traffic for one of the service classes. Each service level pipe provides a particular level of network performance. Integrated services and differentiated services are each supported by one service level pipe. Another service level pipe is used for best effort traffic so that SAAM is backward compatible with legacy networks.

For a network management protocol like SAAM, control traffic management is a critical issue. To support real-time traffic, a network management protocol needs to check the status of the links, routers and flows at short intervals. This enables the network management protocol to adapt to changes in network conditions quickly and maintain the QoS support to real-time traffic flows. SAAM uses a separate service level pipe for *control traffic* which is assigned the highest priority.

1. QoS Parameter Measurements

Each service level is implemented as a queue. Currently these queues are first in first out (FIFO) queues. For each service level queue, performance metrics of interest are: service utilization, average packet delay, and packet loss rate.

These metrics are measured at periods to find out the performance of each SLP. Each metric value is normalized with the previous measurement value. The general formula to calculate a metric is:

$$Metric = \alpha * CurrentMeasurement + (1 - \alpha) * Metric \tag{3.1}$$

a. Utilization

The utilization of a service level queue is calculated by recording the total time that the pipe is busy, i.e. has at least one packet in a measurement period. In busy time calculation we are concerned with two transitions of the queue status: the transition from idle (empty) to busy (populated) and the one from busy to idle. Before a packet is queued, the status of the service level queue is checked. If the queue is empty then the

queue becomes busy and the busy timer is started as soon as the packet is enqueued. If the queue is not empty, meaning it is already busy, there is no need to adjust the timer. Likewise right after a packet is de-queued, the status of the service level queue is checked. If the queue becomes empty then the busy timer stops. If the queue status doesn't change, the timer continues recording the busy time.

Figure 3.2 illustrates the calculation of the utilization calculation of a service level queue. Initially the queue is empty. When a packet arrives at t_1, the queue becomes busy, and the busy timer starts. When the service level queue becomes empty again at t_2, busy timer stops, and the time difference t_2-t_1 is added to the total busy time.

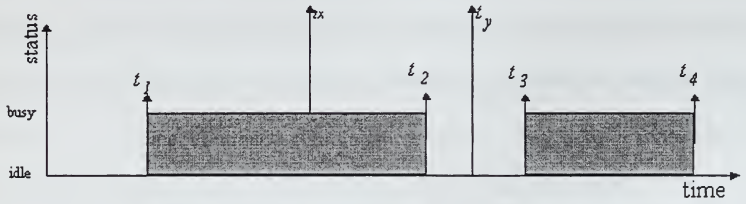


Figure 3. 2. Utilization in Service Level Queue.

The period that the utilization is calculated can end at two places. If the period ends at t_x the total busy time is t_x-t_1, and in this case the busy start time is stamped with t_x, since the queue is still busy. The period may end at some place like t_y where the queue is idle (empty). This time, the total busy time is the sum of previous busy times, t_2-t_1 in this example.

After finding out the *total busy time*, first the *calculated utilization* is calculated by taking the percentage of the busy time in measurement period. Then the *utilization* of the SLP can be calculated as follows.

The idea behind calculating the utility as the percentage of busy time at a specific period can be explained as follows: the SAAM server makes reservations along a path with associated QoS flow parameters specified by the requesting host application. Theoretically, if the bandwidth allocation is 40% at a service level queue then the service level queue is expected to be busy 40% of the time. If the service level queue is busy 80% of the time, 80% of the service level queue bandwidth is used.

b. Delay

Delay is the waiting time a packet experiences in a service level queue. When the packet is en-queued, it is time stamped. And when it is de-queue, another time stamp is taken. The difference between *de-queue time* and *en-queue time* is the *delay* experienced by that packet.

Reported delay is the average delay for the service level queue. To calculate average delay, queued packets are counted in a measurement period. The *previous average delay* is used to normalize the average delay. The average delay is calculated using the following formula.

$$AverageDelay = [AverageDelay * (n-1) + X]/n \qquad (3.2)$$

In Equation 3.2, n is the packet count and X is the delay experienced by the packet. For example, at start of a measurement period the average delay is 0. The first

packet is queued, and it leaves the queue 30 ms later. The average delay is *(0*0 + 30)/1=30* ms. The second packet comes and it leaves the queue 40 ms later. The reported delay is now: *(30*(2-1)+40)/2 = 35* ms.

c. Loss Rate

For loss rate calculation, during each measurement period, the number of packets that have arrived and the number of packets that are dropped in the period are counted. When a packet is about to be en-queued the capacity of the queue is checked. If the packet can not be queued because the maximum queue size is reached, it is dropped. Every packet, including the dropped ones, increments the packet count.

At the end of the measurement period, the percentage of the loss count is calculated first. This percentage is the *observed loss rate* for the period. Since history of the loss rate affects the *current loss rate*, normalization of the observed value with the previous loss rate is required. A normalization value $\lambda\ (0<=\lambda<=1)$ is used to calculate the current loss rate. The formula to calculate the loss rate is as follows:

$$LossRate = \lambda * ObservedLossRate + (1 - \lambda) * LossRate \tag{3.3}$$

2. LSA Data Collection at SLP

Data collection is done at each queue. From the Figure 3.3, data is collected to calculate average delay, loss rate, and utilization of the queue.

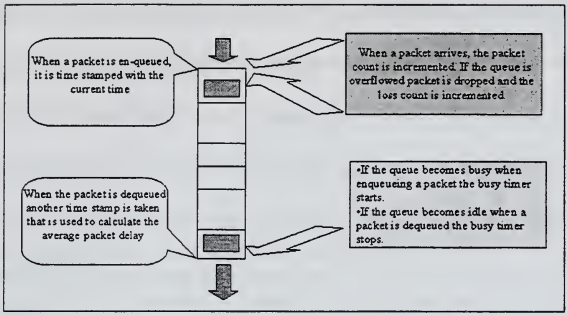

When a packet is en-queued, it is time stamped with the current time
When a packet arrives, the packet count is incremented. If the queue is overflowed packet is dropped and the loss count is incremented.
When the packet is dequeued another time stamp is taken that is used to calculate the average packet delay
•If the queue becomes busy when enqueueing a packet the busy timer starts.
•If the queue becomes idle when a packet is dequeued the busy timer stops.

Figure 3. 3. Collecting Data from SLP.

Each data collection is done by a separate sub module. So different measurement periods can be used for metric calculations.

Metric calculation modules wake up at the end of their measurement periods [Figure 3.3]. They calculate the their metrics using the collected data. Results of the calculation are stored to be presented to other modules when requested.

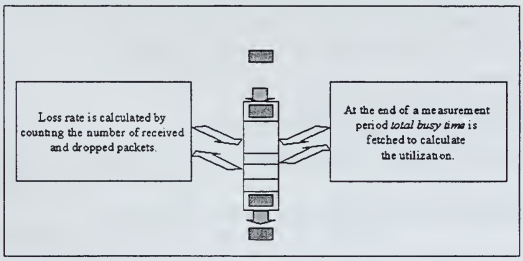

Loss rate is calculated by counting the number of received and dropped packets.
At the end of a measurement period *total busy time* is fetched to calculate the utilization.

Figure 3. 4. Loss Rate and Utilization Modules.

3. Service Level Pipe State Advertisement

The measured QoS parameters of a service level pipe are combined into a *Service Level Pipe State Advertisement (SLP-SA)* message. The SLP-SA message fields, and the number of bytes used to represent each field, are shown in Figure 3.4.

1	2	2	2
SLP Number	Utilization	Delay	Loss Rate

Figure 3. 5. Format of Service Level Pipe State Advertisement (SLP-SA).

- *SLP Number* is the id of the service level pipe. Currently there are four service level pipes. They have ids ranging from 0 to 3.
- *Utilization* is represented by a two-byte integer. Utilization is represented at a granularity of 0.1. The range of the values for this field is 0-1000. 0 means 0% *utilization*, and 1000 meaning 100% *utilization*. For example, a value of 654 means 65.4% utilization.
- *Delay* is represented by a two-byte integer. The unit is milliseconds. The maximum delay is $2^{15} = 32768$ ms.
- *Loss rate* is represented by a two-byte integer. The valid values for loss rate are between 0 and 10000 with a granularity of 0.01. For example 7654 would mean 76.54% loss rate for that service level pipe.

A module collects metric calculation data from a SLP, and combines them into a SLPLSA [Figure 3.6].

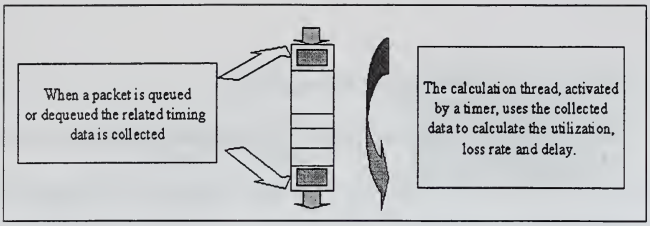

When a packet is queued or dequeued the related timing data is collected
The calculation thread, activated by a timer, uses the collected data to calculate the utilization, loss rate and delay.

Figure 3. 6. Forming SLP-SA.

C. INTERFACE STATE ADVERTISEMENT

The link state of an interface must include states of all its service level pipes. So the *Interface State Advertisement (InterfaceSA)* is the combination of the SLP-SA of the service level pipes. *InterfaceSA* message parameters and the byte lengths are given in Figure 3.7.

1	16	4	1	# of SLP * SLP-SA.length
LSA Type	Interface Id	Bandwidth	# SLP	SLP-SAs

Figure 3. 7. Interface State Advertisement (InterfaceSA).

- *LSA Type* defines the type of this InterfaceSA. Three values are defined so far:
  - Type 0: This is an *Update*. The InterfaceSA carries information that will be used by the SAAM server to update the QoS parameters of the interface in the PIB.
  - Type 1: This is an *Add*. This type is only used when the router has a new interface installed. The SAAM server adds interface and link state data to the PIB.

- Type 2: This is a *Remove*. A router creates this type of LSA when one of its interfaces fails. The SAAM server needs this type to promptly delete those *paths* that use the failing interface from the PIB. Currently the flows using that interface are not rerouted. It is left as future work.

- *Interface id* is the Ipv6Address of the interface. It is used by the SAAM server to identify the interface.
- *Bandwidth* is the total bandwidth of the link associated with the interface in Kbps.
- *Number of SLPs* specifies how many SLP-SAs are sent in the InterfaceSA.
- SLP-SAs are concatenated at the end of the InterfaceSA.

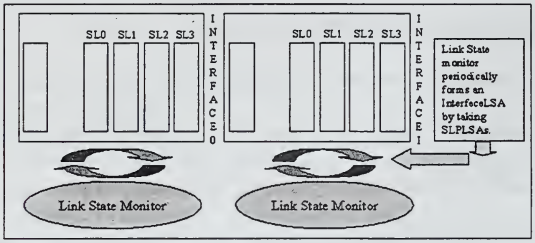


Figure 3. 8. Forming Interface State Advertisement.

Figure 3.8 depicts the forming of the InterfaceSA. Every interface has a module that periodically collects performance data (SLP-SA) from the SLPs. The data combine into an InterfaceSA that describes the performance of the interface.

D. LINK STATE ADVERTISEMENT

The link state of a router is combination of performance of its interfaces. So a *Link State Advertisement (LSA)* message is composed of InterfaceSAs. LSA message format is illustrated in Figure 3.9.

1	16	1	
Mes. Type	Router Id	# Interface	# of Interfaces * InterfaceLSAs

Figure 3. 9. Link State Advertisement (LSA).

- *Message type* distinguishes the message as an LSA. The type for LSA is 12.
- *Router id* is an Ipv6Address. It is the biggest of the interface IP numbers of the router. The *router id* is selected as follows:
    - Starting from the most significant byte, when one of the byte of one interface is bigger than that of any other interface, IP number of the interface is taken as the router id. Selection continues until the last bytes. Since each IP number is unique, one of the interface IP numbers will be bigger than the others.
- *Number of interfaces* shows how many interfaces the router has.

LSA messages are created periodically by a generation module. This module collects link states of each interface and combines them into an LSA message. Figure 3.10 depicts the generation of LSA message. Routers piggyback their LSA messages to the UCM message, using the ability of the SAAM Message's concatenation capability.

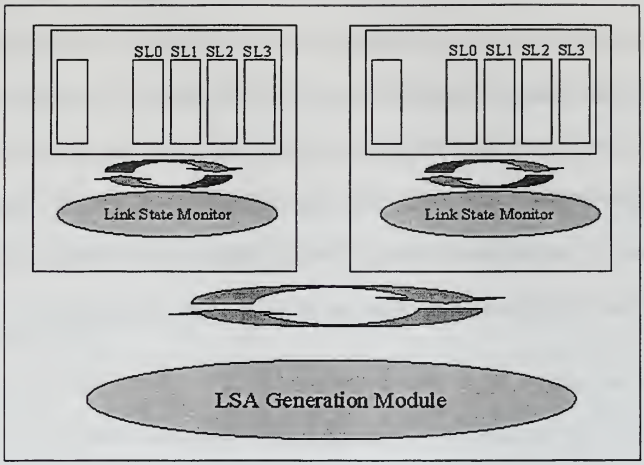


Figure 3. 10. Generation of Link State Advertisement.

E. INTERFACE FAILURE

A SAAM server makes flow reservations for real-time traffic. Real time traffic requires QoS support. The SAAM server needs to react to link failures in a very short time to meet the QoS requirements of the real time traffic flows. Interface failure is simulated by sending InterfaceFailure [Figure 3.11] message to a router. This message is not a part of the SAAM. It is used just for simulation purposes.

1	16
Message Type	IPv6Address

Figure 3. 11. InterfaceFailure Message Format.

- Message type value is 127.
- IPv6Address is the address of the failing interface

When a router receives an InterfaceFailure message, it finds the interface and changes the status to *down*. Interface failure is detected by the LSA generation module. The LSA generation module always checks the status of the interfaces while creating LSA messages. When it detects an interface failure, it will report to the server by a triggered LSA. The server, after receiving the triggered LSA, updates its PIB by removing the paths that contain a SLP of the failed interface. The server should reroute the flows using those paths. Rerouting of flows is left as future work.

IV. IMPLEMENTATION

An implementation of the LSA mechanism has been carried out with the Java programming language. This implementation is a part of the most recent SAAM system prototype. The code is written in a modular way. By making the code modular, it was possible to debug and verify a small number of functionalities at a time. For efficiency, independent tasks are performed with separate threads whenever possible.

A. LSA GENERATION

Every node (router or server) in a SAAM region performs LSA generation tasks. The server has the additional responsibility of processing LSA messages. The following Java classes are used for LSA generation:

1. PriorityQueue

PriorityQueue replaces the previous *queue* implementation which does not have the packet performance data collecting primitives. *PriorityQueue* has built-in data collecting functionalities. *PriorityQueue* has a double-linked list based data structure. It has all the functionalities of a first-in-first-out queue. Figure 4.1 shows the methods of *PriorityQueue*.

PriorityQueue keeps track of the number of packets currently queued by using the *size* variable. The size of the queue can grow to *maximum queue size*, a variable used to define capability of the queue. If no parameter is given, the default constructor creates an instance with the *default maximum size* which is 200. Both constructors initialize those

variables used to calculate the average packet delay, service level utilization, and packet loss rate.

Each service level pipe is implemented by a *PriorityQueue* object. Service level queues collect service level performance data and perform calculations to determine the value of each metric.

When a packet is queued, *size* is compared with the *maximum queue size*. If *maximum queue size* is reached, the packet is dropped, and the loss count and packet count are incremented. Otherwise, just the packet count and the queue size are incremented.

If the queue was empty before a packet is queued, it becomes busy. The current time is recorded as the busy start time. A *QueueItem* object is created that holds the packet and the arrival time stamp of the packet. This *QueueItem* is inserted to the end of the queue.

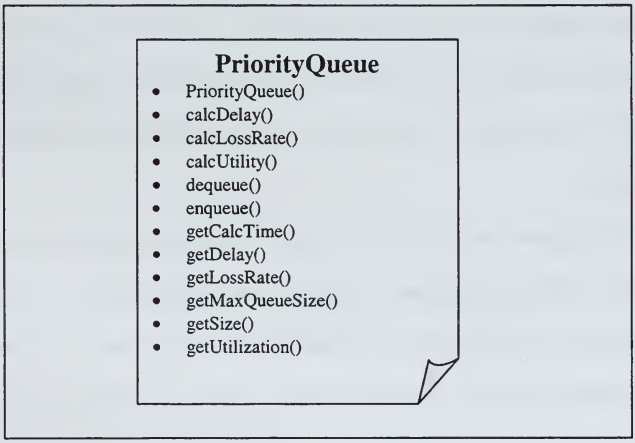


Figure 4. 1. saam.util.PriorityQueue

While a packet is dequeued, the queueing delay of the packet is calculated. The value is the difference between the arrival time and the departure time of the packet. Packet queueing delays are used to calculate the average packet delay.

To find out the service level utilization, the total busy time in the current measurement interval must be determined. The start and end times of each *busy period* are recorded to find out the length of the *busy period*. A *busy period* starts when a packet is put into an empty queue. The *busy period* ends when a service level queue becomes empty when a packet is dequeued. The sum of the *busy periods* gives the *total busy time* in a measurement period.

Service level utilization is calculated by using the *total busy time*. A timer periodically creates an action sets the flag, a boolean variable, that starts the service level utilization calculation. A separate thread performs the utilization calculation, and then lowers the flag and enters a *wait state*. For utilization calculation details please refer to Chapter 3.

A similar approach is taken for the packet loss rate calculation. A separate thread calculates the loss rate of the service level queue. A second timer periodically invokes the loss rate thread. After invocation, thread calculates the loss rate by computing the percentage of packets that are dropped in the current period and normalizing the result by the previous measurement interval.

2. Service Level Pipe State Advertisement (SLP-SA)

SLP-SA is the class that manages the state of performance of a service level pipe [Figure 4.2]. Performance data taken from a service level pipe together with the *SLP number* are combined into an *SLP-SA*. *SLP-SA* has four data members: *SLP Number*, service level *utilization*, *packet delay*, and *packet loss rate*. One constructor of the class takes three parameters, one for each of these values, and creates an SLP-SA. The other constructor takes these data in a byte array and constructs the SLP-SA by extracting data members from the byte array.

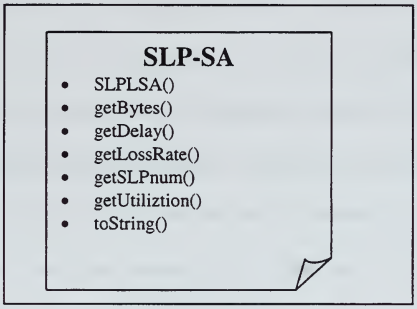


Figure 4. 2. saam.message.SLP-SA

For each data member, there is a *get* method to retrieve the value. The values of all data members can be put into a byte array and retrieved using the *getBytes()* method.

3. Interface State Advertisement (InterfaceSA)

InterfaceSA is the class that manages the performance status of the interface [Figure 4.3]. It is created by the *LinkStateMonitor* of each interface. It has three constructors. The first one takes a byte array as parameter and constructs the class by

extracting all data members from the array. The second takes an *IPv6Address* and an integer number for the bandwidth of the interface. The third takes an *IPv6Address*, an integer bandwidth, and a byte for type of *InterfaceSA*. The *InterfaceSA* packet format is shown in Figure 3.6.

InterfaceSA can be one of three types: *remove, add*, and *update*. Default type is *update*. When an interface is to be added to the PIB, an *InterfaceSA* of type *add* is created. Likewise when an interface is to be removed an *InterafceSA* of type *remove* is created.

SLP-SAs can be added to an *InterfaceSA* either one by one or by a vector of *SLP-SAs*. Each added *SLP-SA* increments the *SlpNumber* value, the variable used to keep track of the total number of *SLP-Sas* in an *InterfaceSA*.

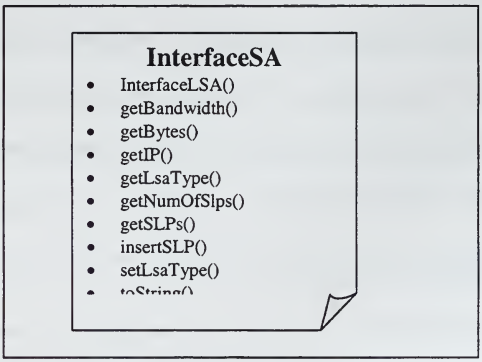


Figure 4. 3. saam.message.InterfaceSA

All data members of *InterfaceSA* can be retrieved using the *getBytes()* method.

4. LinkStateMonitor

LinkStateMonitor is the class, that generates *InterfaceSA* messages. Each interface has a *LinkStateMonitor* that monitors the performance of the service level pipes of the interface.

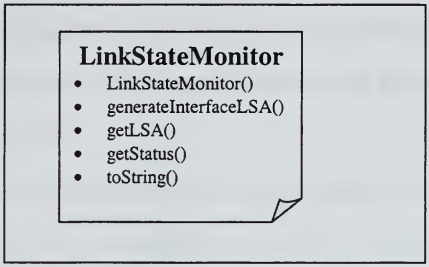


Figure 4. 4. saam.residentagent.LinkStatemonitor

LinkStateMonitor generates an *InterfaceSA* when there is a *UCM* packet ready to be sent to the server or when the *LsaGenerator* requests one. It uses the *generateInterfaceLSA()* method and reads the *SLP-SA* of each service level pipe. It then passes the InterfaceLSA to *LsaGenerator*.

5. LinkStateAdvertisement

LinkStateAdvertisement is the class that manages the performance status of a router. *LinkStateAdvertisement* extends *saam.message.Message* and its *message type* is 12.

LinkStateAdvertisement has two constructors. The first one takes the *IPv6Address router id*, which is the biggest interface address of the router. The second constructor

creates a byte array representation of the data. It is used by the server to reconstruct the *LinkStateAdvertisement* packet from the byte array.

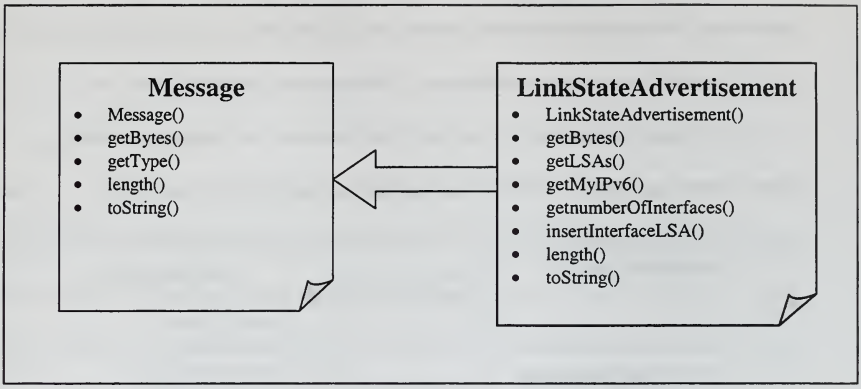


Figure 4. 5. saam.message.LinkStateAdvertisement

LinkStateAdvertisement contains the *InterfaceSAs* of all the interfaces of a router. It keeps them in a vector. *InterfaceSA* messages can be added to a *LinkStateAdvertisement* message individually or together as a vector. When an *InterfaceSA* is added *numOfinterface*, the variable used to keep track of the total number of *InterfaceSA* messages in a *LinkStateAdvertisement*, is updated. The *numOfinterface* variable is used when constructing a *LinkStateAdvertisement* message from a byte array by the server.

6. LsaGenerator

LsaGenerator, as its name implies, is the class that generates *LinkStateAdvertisement* messages. It is instantiated by *ControlExecutive* of a router.

LsaGenerator starts generating LSAs after the interfaces are initialized by a *DemoHello* message.

LsaGenerator has two constructors. One of them takes *ControlExecutive* as a parameter. It uses the default LSA generation interval, which is ten seconds. The other takes *ControlExecutive* and an integer for LSA *generation interval.*

A timer periodically invokes *LsaGenerator* by setting a boolean flag. *LsaGenerator* generates an LSA by calling the *performLSACycle()* method.

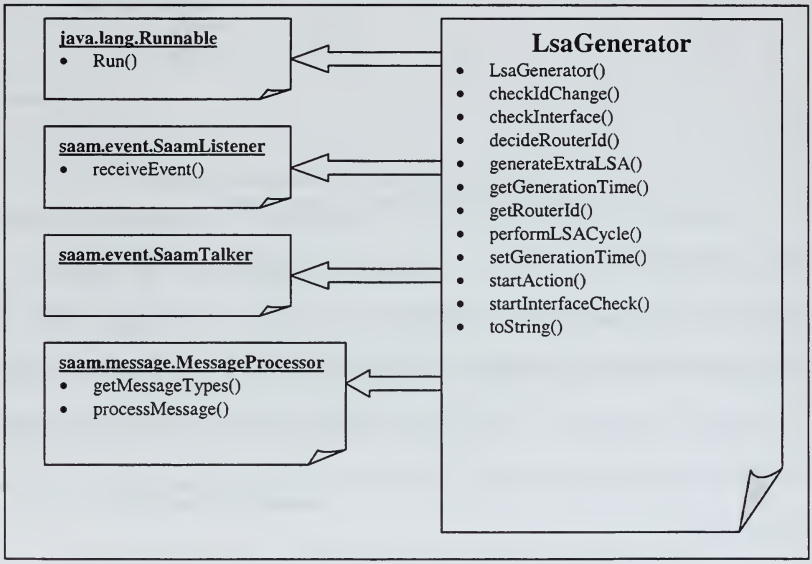


Figure 4. 6. saam.residentagent.router.LsaGenerator

While generating an LSA, *LsaGenerator* first checks the status of each interface. If the interface is *up*, it takes an *InterfaceSA* from the interface. It combines every

InterfaceSA and adds other LSA fields to form an LSA message. The LSA message is passed to *AutoControlExecutive* to be concatenated to a *UCM* message. When *LsaGenerator* discovers that an interface is down, it creates two LSA messages of the *remove* type for the down interface. These LSA messages are sent to the p*rimary* and *backup* servers using *sendLSA()* method of the *ControlExecutive*.

The reason to send two consecutive LSAs is to be sure that the server receives at least one of the LSAs. Since the down interface could be the one that was used to reach to the server. If the LSA message is just sent once, it may not reach the server. By auto-configuration, the SAAM control channel is refreshed at every cycle. In the next cycle, router will learn the new way to reach the server. So the server will receive the second LSA even if the first one got lost.

Another thread is used in the *LsaGenerator* class to check the interfaces. The check thread is active at every two seconds. After each check it goes to *wait state*. There is a second timer, which triggers the checks. This timer periodically raises a boolean flag to wake up the check thread. The check thread examines the status of each interface. If no interface is down, it lowers the flag and goes back to the *wait state*. If it detects a *down* interface, it generates and sends two consecutive LSA messages as described above. Therefore, an interface failure will be detected in at most two seconds.

B. LSA PROCESSING

1. ServerAgent

The *ServerAgent* class is the receiver of all server-bound messages. It uses a *Server* object to process these messages. In the previous implementation, *ServerAgent* would receive *Hello* messages from all the routers in the region. *Server* then would use these *Hello* messages to build up the PIB.

With the addition of Link State Advertisement model, the topology information is carried by LSA messages. Since the need for *Hello* messages is gone, the portion of code handling *Hello* messages is removed from *processMessage()* method.

2. Server

Server builds up the PIB and maintains it by processing LSA messages received from routers in the region.

A new table, called *RouterLookUp,* is created to map the new *IPv6Address router id* with the *integer router id* used by the previous implementation. This table provides a link between the two implementations.

The following methods are added to the Server class:

a. void processLSA(LinkStateAdvertisement lsa)

LSA messages received by *ServerAgent* are passed to this method for processing. This method first extracts the *Ipv6Address router id* from the LSA message. Using the *RouterLookUp* table it tries to learn the integer id of the router.

If the *IPv6Address router id* is not in the lookup table, the server will try finding the integer id by using all interface addresses learned from the LSA message. It will send these addresses to the *doesRouterExist()* method. The *doesRouterExist()* method will return an integer value. A return value of *ROUTERNOTINPIB* indicates that the server has received an LSA from a router for the first time and the *newRouter* flag is set to true. A new *integer router id* is assigned to this router and *RouterLookUp* table is updated.

Using the *LinkStateAdvertisement's getLSAs()* method InterfaceSA messages are retrieved in a vector. Starting from the first element in the vector, the server processes each *InterfaceSA* messages.

If the type of the *InterfaceSA* is *add*, the interface is added to the PIB. With the addition of a new interface, the *router id* of a router may change. If there is id change, the *RouterLookUp* table will be updated. After adding a interface, the link state data is updated and *findAllPossiblePaths()* method is used to add new paths to the *paths* table. If the *InterfaceSA* type is *update*, the *updatePIB()* method is called to update the related interface state data in the PIB. If the type is *remove*, the server checks if the interface is in the PIB. If it is not in the PIB, nothing will be done. If it is in the PIB, the *removeInterfaceFromPIB()* method will be called to remove the interface and those *paths* using that interface. If the removed interface address is the *router id* of a router a new *router id* is calculated for that router.

When a new router is added to the PIB, all paths are updated by the *findAllPossiblePaths()* and the effective QoS is calculated for all paths.

b. void checkAndAdd(int nodeId, InterfaceSA curLsa)

To add a new interface this method is called. This method checks if the node with *nodeId* has an interface with the IPv6Address specified by *curLsa*. If the interface is not already in the PIB, it is added to the PIB. Also all *SLPs* of the interface are added to the PIB. The link state data of the interface initialized based on *curLsa*.

c. void updatePIB(int nodeId, InterfaceSA curLsa)

If the *InterfaceSA* sent by a router is an *update*, it is sent to this method. All *SLP-SAs* are taken from the *curLsa* as a vector. The node to be updated is found from the *nodes* table using *nodeId* and the interface is found from the *IPv6Address* of *curLsa*.

d. void removeInterfaceFromPIB(int nodeId,InterfaceLSA curLsa)

This method removes an interface and those paths using a *SLP* of the interface. First it calls *removePathsTraversingInterface()* that removes the paths using the *IPv6Address* address of the interface as key. Then it calls *removeLinkFromPIB()* that removes the *link* associated with the interface. Finally, it calls the *removeInterfaceFromNode()* method to remove the interface from the *nodes* table.

e. void removePathsTraversingInterface(IPv6Address ip)

This method deletes all paths that use interface. By using the *getAllPathIdsThatTraverseSLP()* method, it gets the ids of paths to be removed in a vector and deletes all paths from the *paths* hash table.

f. void removeLinkFromPIB(IPv6Address ip)

After deleting all paths traversing interface, this method is called to remove the associated link from the *links* hash table.

g. void removeInterfaceFromNode(IPv6Address ip)

This method deletes the interface *ip* from the *nodes* hash table.

h. void checkRouterId(IPv6Address routerId,Vector iLsaVector)

If one of the InterfaceLSA messages is of type *remove* or *add*, this method is called. It checks if the *router id* requires change or not. It uses the same algorithm to calculate the router id as the routers so that the *router ids* are same at the router and the server. New LSAs received from that router will have this new router id because the same procedure was done at the router, too. Currently the *addition* of an interface to a running router is not implemented yet.

C. SIMULATION OF INTERFACE FAILURE

When an interface fails, server should know the failure and take necessary steps for the affected flows. Routers inform the server about the failure of an interface with LSA messages.

The interface that failed may be the one used to reach the server. In this case, if the LSA is sent once, it may not reach the server. To make sure that server receives the failure information, the LSA carrying this information is sent a second time. Since auto-

configuration will refresh the control channel at the next configuration cycle, router will learn the new way to reach the server bypassing the failed interface.

To simulate the failure of an interface, an *InterfaceFailure* message [Figure 4.7] is sent from the *DemoStation* to a router. This message has an *IPv6Address* as data member, which is the address of the failed interface.

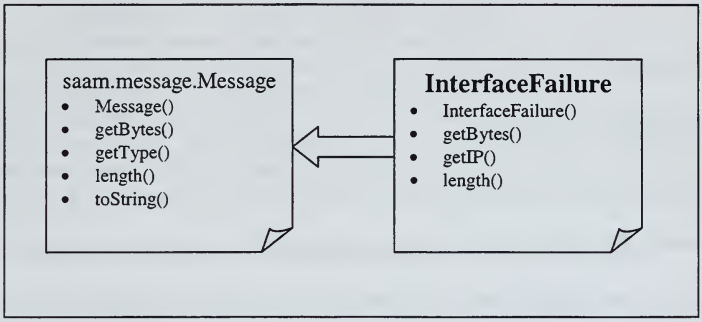


Figure 4. 7. saam.message.InterfaceFailure

The message processor of the failure message is *ControlExecutive*. When *ControlExecutive* receives an *InterfaceFailure* message, it compares the *IPv6Address* of each interface with the address in the message. If *ControlExecutive* finds a match, it sets the interface *status* variable of that interface to false, which means the interface is *down*. The *status* of an interface is checked every time a packet enters the interface. If the status is down, the packet along with those already in queue are dropped.

The failure of the interface will be detected by *LsaGenerator*. A part of the LsaGenerator checks the status of the interfaces in short periods. When it detects that an

interface is down, it creates a LinkStateAdvertisement and sends it to every server in the region by using the *sendLSA()* method of the *ControlExecutive*.

THIS PAGE INTENTIONALY LEFT BLANK

V. INTEGRATION AND TESTS

By the completion of the auto-configuration of control channel, routers in a SAAM region send LSA messages by concatenating them to UCM (Upward Configuration Messages). Each server learns its children and builds up its PIB by the concatenated LSA messages.

A. INTEGRATION

Servers and routers have some differences concerning UCM and LSA messages. A server sends LSA messages to itself and to other servers. It processes both LSA and UCM messages. On the other hand, routers only process UCM message of a child, from which a PN (Parent Notification Message) has arrived. To meet the different requirements of servers and routers, differentiation is made at PacketFactory, which is the common place for both message types.

Each router creates a *ServerInformation* class for each server, where it keeps track of the information about the server. ServerInformation class is chosen to store the concatenated LSA messages on the way to a server.

Figure 5.1 illustrates the UCM message handling at a router. When a UCM message group, UCM message with concatenated LSA messages, arrives to a router, it is forwarded from PacketFactory to AutoControlExecutive. AutoControlExecutive extracts the UCM message from the message group, and learns the flow id of the server, which the packet is destined for. Using the flow id of the server as key, associated ServerInformation is learned. The concatenated LSA messages and the number of LSA

messages are passed to the ServerInformation of the server. AutoControlExecutive continues with the processing of the UCM message.

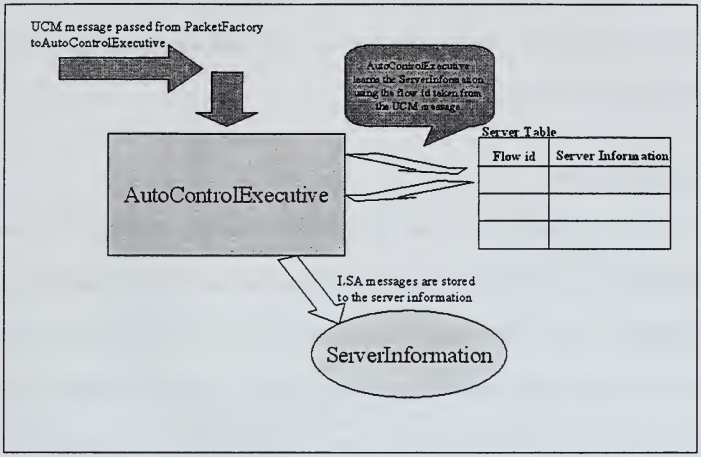

UCM message passed from PacketFactory toAutoControlExecutive
AutoControlExecutive learns the ServerInformation using the flow id taken from the UCM message
Server Table
Flow id
Server Information
AutoControlExecutive
I.SA messages are stored to the server information
ServerInformation

Figure 5. 1. UCM Arrival and Storing to ServerInformation.

Router prepares its UCM message by using the received UCM messages. When the router UCM message is ready, LSA message of the router is concatenated to the message. Received LSA messages are taken from the ServerInformation and concatenated to the UCM message group. While concatenating messages, number of messages is updated.

The server is the processor of the UCM messages. When a UCM message group arrives to a server, messages are extracted from the group and processed individually depending on the message type. PacketFactory check if it is a server, when it receives a UCM message by calling the *ControlExecutive getIsServer()* method.

B. TESTS

LSA concept is tested in a three-computer test network. Figure 5.2 depicts the topology of the test network. All computers have 166 MHz CPU speed and connected by a 10 MBPS Ethernet LAN.

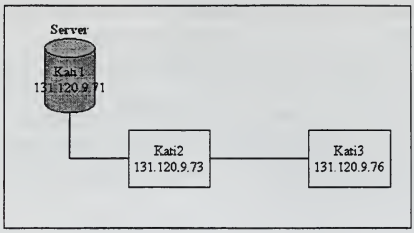


Figure 5. 2. Test Topology.

In the current implementation the PIB is built up from the scratch when the server learns that there is a new router. This takes extra time and inefficient. Another point is the emulated environment. Ipv6 packets are sent using an emulated network stack. Layers of the network are simulated in software, which is another reason why the PIB build up is slow. First element added to the PIB is the server itself. The server sends LSA messages to itself. Routers are added to the PIB when the server receives the LSA messages that are concatenated to a UCM message.

Table 1 shows the times of the PIB build up when GUI comments are enabled:

	Low	High
Kati1(Server)	75 milliseconds	90 milliseconds
Kati2	5 seconds	6.5 seconds
Kati3	26 seconds	29 seconds

Table 1. PIB Build Times.

These results improve at a certain level when the GUI comments are disabled. But the overall performance still lacks the speed and efficiency. Table 2 shows the test results when the GUI comments are disabled.

	Lowest	Highest
Kati1(Server)	60 milliseconds	80 milliseconds
Kati2	4.5 seconds	6 seconds
Kati3	22 seconds	27 seconds

Table 2. PIB Build Without GUI Comments.

VI. CONCLUSIONS

A. SUMMARY

In this thesis I implemented a model for generating and processing *Link State Advertisements* in the SAAM architecture. I implemented the model in the Java programming language. I debugged and improved the old threading model of the previous SAAM prototype. Further, I integrated my new design to those of four other colleagues. I implemented each sub-model in a separate thread for performance and parallelism. This work substantially improved and added to the overall functionality of SAAM.

B. LESSONS LEARNED

The main focus of this study is to develop a model that delivers link state performance data of the routers to the server in the SAAM architecture in a timely manner. Integrating the model with the previous version of SAAM took more time than the development of the model. In most cases, a new functionality can be implemented as a separate module to make the design more modular. But some code modifications and additions to existing modules were inevitable. The biggest difficulty came from the lack of knowledge about of the Java programming language the *thread support*. Sometimes a thread synchronization problem took a week of study to debug and correct.

Another lesson is the need to coordinate between the team members of the current development group. Each functionality of SAAM is developed as a thesis study by one of

the team members. The modifications and improvements made one member impact the work of other team members. This taught us the need to develop a software development paradigm for developing a large system like SAAM.

C. FUTURE WORK

1. Improving the PIB Path Processing

Currently the PIB is built from scratch when the server learns a new router in the region. A more efficient algorithm could be deployed that adds new paths to the current PIB, rather than rebuilding the whole PIB.

2. Rerouting of the Flows if an Interface Fails

When an interface fails, the SAAM server should reroute the flows that are affected. The server should also update the PIB by deactivating the paths that include a SLP of the failed interface. This feature is currently not available.

3. Securing SAAM

SAAM server is the central authority managing the region. It must be secured to prevent hackers from disrupting network services. Control traffic must be authenticated to prevent illegal alterations to PIB by malicious attacks.

APPENDIX A. PRIORITYQUEUE CLASS SOURCE CODE

```
package saam.util;

import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.text.DecimalFormat;

import saam.message.SLPLSA;


/**
 * this class will represent a priority queue for the saam packets
 * smaller the value higher the priority
 */
public class PriorityQueue extends FIFOQueue{

  /**
   * default size of the queue
   */
  public static final int DEFAULT_SIZE=200;

  public static final byte MAX_UTIL = 100;

  public static final short MAX_DELAY = Short.MAX_VALUE;

  public static final short MAX_LOSS = 10000;

  /**
   * int representing milliseconds to recalculate the values of
lossRate,delay
   * and utilization    - 5 minutes
   */
  public static final int TIME_TO_RECALCULATE=2000;

  public static final double ALFA_FOR_LOSSRATE = 0.7d;

  private int calculationTime;

  /**
   * max size if the queue
   */
  private int maxSize;

  private long packetCount;

  private long lossCount;

  private Timer timer;

  private double alfa;

  private double currentLossRate=0.0d;
  private boolean lossStart = false;
  private Object lossLock=new Object();
```

```
private double previousLossRate=0.0d;

private double reportRate=0.0d;

private long currentPacketDelay;

private double currentDelayAve;

private double previousDelayAve;

private double utility=0.0d;
private double preUtility=0.0d;
private long busyTime=0;
private long busyStartTime;
private double utilRatio = 0.8d;
public static final int DEF_UTIL_CALC_PERIOD=25000;
int utilPeriod = DEF_UTIL_CALC_PERIOD;
private Timer utilTimer;
private boolean utilStart=false;

private Object theLock=new Object();

private Object utilityLock = new Object();

/**
 * current size of the queue
 */
private int size;

private QueueItem first,last,current;

/**
 * parameterless constructor of the class
 */
public PriorityQueue() {
  maxSize = this.DEFAULT_SIZE;
  size = 0;
  packetCount=0;
  lossCount=0;
  first = last = current = null;
  calculationTime = TIME_TO_RECALCULATE;

  Runnable lossRunner = new Runnable(){
      public void run(){
        while(true){
          try{
            synchronized(lossLock){
              while(!lossStart){
                lossLock.wait();
              }
            }
          }catch(InterruptedException ie){
            System.err.println(ie);
          }
          calcLossRate();
        }//while
      }
```

```
    };
  Thread lossThread = new Thread(lossRunner,"LoosRate");
  lossThread.start();

  timer = new Timer(calculationTime,new ActionListener(){
    public void actionPerformed(ActionEvent event){
      synchronized(lossLock){
        lossStart = true;
        lossLock.notify();
      }
    }
    });
  timer.start();

  Runnable utilRunner = new Runnable(){
      public void run(){
        while(true){
          try{
            synchronized(utilityLock){
              while(!utilStart){
                utilityLock.wait();
              }
            }catch(InterruptedException ie){
              System.err.println(ie);
            }
            calcUtility();
          }//while
      }
    };
  Thread utilThread = new Thread(utilRunner,"Utilization");
  utilThread.start();
  utilTimer=new Timer(utilPeriod,new ActionListener(){
      public void actionPerformed(ActionEvent event){
        synchronized(utilityLock){
          utilStart = true;
          utilityLock.notify();
        }
      }
    });
  utilTimer.start();
  alfa = ALFA_FOR_LOSSRATE;
}//end constructor

/**
 * parameter constructor
 * @param queuesize int size of the queue
 */
public PriorityQueue(int queuesize){
  maxSize = queuesize;
  size=0;
  packetCount=0;
  lossCount=0;
  calculationTime = TIME_TO_RECALCULATE;
  timer = new Timer(calculationTime,new ActionListener(){
    public void actionPerformed(ActionEvent event){
      calcLossRate();
```

```
      }
      });
    timer.start();
    utilTimer=new Timer(utilPeriod,new ActionListener(){
        public void actionPerformed(ActionEvent event){
          calcUtility();
        }
      });
    utilTimer.start();
    alfa = ALFA_FOR_LOSSRATE;
  }//end constructor


  private void calcUtility(){
    if(!isEmpty()){ //there are packets in the queue
      long currentTime = System.currentTimeMillis();
      busyTime = busyTime+(currentTime -busyStartTime);
      busyStartTime = currentTime;
    }

    if(busyTime>utilPeriod){
      busyTime = utilPeriod;
    }

    utility = (busyTime/(double)utilPeriod)*100;
    utility = (utilRatio * utility) + (1-utilRatio)*preUtility;
    preUtility = utility;
    busyTime=0;
    DecimalFormat df = new DecimalFormat("##.###");
    String dfstr=df.format(utility);
    //System.out.println("Utiliy is "+dfstr);
    utilStart = false;
  }

  public int getCalcTime(){
    return calculationTime;
  }

  public void setCalcTime(int newTime){
    //do not allow a time smaller than 1 sec
    if(newTime<1000){
      System.out.println("The smallest value is 1 sec.(1000).");
      return;
    }
    calculationTime = newTime;
  }

  public double getAlfa(){
    return alfa;
  }

  public void setAlfa(double newAlfa){
    //alfa is a value between 0 and 1
    if(newAlfa<0.0 || newAlfa>1.0){
      System.out.println("New alfa value must be between 0 and 1. Alfa
is not set.");
      return;
```

```
    }
    alfa = newAlfa;
  }

  /**
   * queuees a new packet depending on the priority of the packet
   * @param packet Object
   * @param priority int priority of the
   */
  public synchronized void enqueue(Object packetArray){

      //if size exeeds the maxSize
      //we need to drop the packet since there is no place in the queue
      if((size+1)>maxSize){
        lossCount++;
        packetCount++;
        return;
      }
      if(isEmpty()){
        //queue was idle
        busyStartTime=System.currentTimeMillis();
      }

      size++;
      packetCount++;

      byte [] packet = (byte[]) packetArray;

      long ts=System.currentTimeMillis();
      QueueItem newItem=new QueueItem(packet,ts);
      if(first==null){
        first = last = newItem;
      }
      else{
        current=first;
        while(current.next!=null){
          current = current.next;
        }//end while
        current.next=newItem;
        newItem.previous=current;
        last=newItem;
      }//end else

  }//end enqueue()



  /**
   * removes the highest priority packet from the queue
   * @return Object
   */
  public synchronized Object dequeue(){

      if(first!=null){
        calcDelay();
        size--;
        if(isEmpty()){
```

```
          busyTime = busyTime +(System.currentTimeMillis()-
busyStartTime);
          System.out.println("Total busy time is "+busyTime);
        }

        current = first;
        if(first.next==null){//item is also the last
          first = last = null;
        }
        else{
          first=current.next;
          first.previous=null;
        }
        return current.data;
      }//end if
      return null;

  }//end method dequeue()


  private void calcLossRate(){
    currentLossRate = 0.0;
    if(packetCount==0){
      packetCount=1;
    }
    currentLossRate = (lossCount/(double)packetCount)*100;
    lossCount = 0;
    packetCount = 0;
    reportRate = (alfa*currentLossRate) + (1-alfa)*previousLossRate;
    previousLossRate = currentLossRate;
    lossStart=false;
  }

  /**
   * returns the loss rate as percentage
   * @return int
   */
  public short getLossRate(){
    return (short)(reportRate/SLPLSA.LOOSRATE_UNIT);
  }//end method getLossRate()

  /**
   * returns the Object the first packet in the queue - highest priority
   * @return Object
   */
  public synchronized Object peek(){
      calcDelay();
      return first.data;
  }//end method peek()

  private void calcDelay(){
    long outTs = System.currentTimeMillis();
    currentPacketDelay = outTs - first.timeStamp;
    currentDelayAve = ((previousDelayAve*(packetCount-1))+
currentPacketDelay)/packetCount;
  }
```

```
/**
 * returns the delay amount of the packet in the queue
 * @return int
 */
public short getDelay(){
  return (short)currentDelayAve;
}//end method getDelay()

/**
 * returns the size of the queue
 * @return int
 */
public int getSize(){
  return size;
}//end mthod getSize()

/**
 * returns true if the queue is empty
 * @return boolean
 */
public boolean isEmpty(){
  return size==0;
}//end isEmpty()

/**
 * returns the int utilization of the queue
 * @return int
 */
public synchronized byte getUtilization(){
  return (byte) (utility/SLPLSA.UTIL_UNIT);
}//end method getUtilization()

/**
 * returns the int max size of the queue
 * @return int
 */
public synchronized int getMaxQueueSize(){
  return this.maxSize;
}//end method getMaxQueueSize()

/**
 * changes the max queue size
 * @param newSize int size to be set
 */
public synchronized void setQueueSize(int newSize){
  //it should atleast be 1
  if(newSize<1){
    return;
  }
  this.maxSize=newSize;
}//end method setQueueSize()


/**
 * return a string representation of the queue
 * @return String
 */
```

```
  public synchronized String toString(){
    return "Queue ";
  }//end method toString()

}//end class priorityqueue


class QueueItem{
  public Object data;
  public QueueItem next,previous;
  public long timeStamp;

  public QueueItem(Object packet,long timeStamp){
    data=packet;
    this.timeStamp=timeStamp;
    next=null;
    previous=null;
  }//end constructor
}//end class QueueItem

//end file PriorityQueue.java
```

APPENDIX B. LINKSTATEMONITOR CLASS SOURCE CODE

```
package saam.residentagent;

import java.util.TooManyListenersException;
import java.util.Vector;
import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

import saam.router.*;
import saam.event.*;
import saam.util.*;
import saam.control.*;
import saam.net.*;
import saam.message.*;
import saam.residentagent.router.LsaGenerator;


public class LinkStateMonitor {

  private SAAMRouterGui gui;
  private Interface parent;
  private int instance;
  private ControlExecutive control;
  private boolean firstTime=true;
  private Object theLock = new Object();
  private boolean started = false;
  private InterfaceLSA myLSA;



  //I plan to use this boolean varialble to simulate the link UP or DOWN
state
  private boolean interfaceState = true;

  public LinkStateMonitor(Interface parent,ControlExecutive controller){
    this.parent=parent;
    instance = parent.getInstanceNumber();
    //gui = new SAAMRouterGui("Link State Monitor "+instance);
    control = controller;

  }//end constructor



  public InterfaceLSA getLSA(){
    generateInterfaceLSA();
    return myLSA;
  }

  /**
   * creates the interfaceLSA for this interface and send it to LSA
generator
   * @return void
   */
```

```
  private synchronized void generateInterfaceLSA(){

    //gui.sendText("genereting interface lsa");
    //first append the header section of the InterfaceLSA
    InterfaceID id = parent.getID();

    if(firstTime){
      myLSA = new
InterfaceLSA(id.getIPv6(),id.getBandwidth(),InterfaceLSA.ADD);
      firstTime = false;
    }
    else{
      myLSA = new InterfaceLSA(id.getIPv6(),id.getBandwidth());
    }

    Vector slpVector = new Vector();

    //for every Service level queue we need to know the servicelevel
data
    for(int i=Interface.SERVICE_LEVEL_QUEUE_START_INDEX;

i<Interface.numberOfQueuesOnThisInterface;i++){

      PriorityQueue sl=null;
      try{
        sl = (PriorityQueue)parent.getQueue(i);
      }catch(Exception e){
        //gui.sendText(e.toString());
      }

      short delay = sl.getDelay();
      short lossRate = sl.getLossRate();
      byte utilization = sl.getUtilization();


      SLPLSA slLSA = new SLPLSA((byte)(i-1),utilization,delay,lossRate);
      slpVector.add(slLSA);
    }//end for

    myLSA.insertSLP(slpVector);

    started = false;

  }//end method generateInterfaceLSA()

  public boolean getStatus(){
    return  interfaceState;
  }

  public String toString(){
    return ("LinkStateMonitor");
  }
}
```

APPENDIX C. LSAGENERATOR CLASS SOURCE CODE

```
package saam.residentagent.router;

import java.net.*;
import java.util.Vector;
import java.util.Enumeration;
import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;


import saam.residentagent.*;
import saam.router.*;
import saam.event.*;
import saam.util.*;
import saam.control.*;
import saam.net.*;
import saam.message.*;

public class LsaGenerator implements SaamListener,
SaamTalker,MessageProcessor,
        ActionListener,Runnable{


  //time between two LSA generations
  public static final int LSA_PERIOD = 10000;

  /**
   * gui of the LSA Generator
   */
  private SAAMRouterGui gui;

  /**
   * reference to the ControlExecutive
   */
  private ControlExecutive controlExec;

  /**
   * id of the router
   */
  private IPv6Address routerId = new IPv6Address();

  private IPv6Address oldId;

  /**
   * numOfInterfaces
   */
  private int numberOfInterfaces=0;

  /**
   *
   */
  private int packetsReceived, packetsSent;

  /**
```

```
   * source and destination ports
   */
  private short sourcePort,destPort;

  /**
   * vector that holds InterfaceIDs
   */
  private Vector interfaceIDs;

  /**
   * @serial
   */
  private String[] messageTypes =
    {};

  private Object myLock = new Object();

  private Timer timer;

  private int timeBetweenLSAGenerations;

  private boolean started=false;

  private Thread runner;

  private LinkStateAdvertisement LSA = null;

  private Vector interfaces;

  private LinkStateMonitor monitor;

  private boolean idChange=false;


/**************************************************************************
****
    this thread will check the status of the Interfaces and generate an
extra
    LSA if the status of the Interface is down or

**************************************************************************
****/
  private Thread intChecker;

  private Timer checkTimer;

  private boolean go=false;

  private Object checkLock=new Object();


  public LsaGenerator(ControlExecutive control){
    gui = new SAAMRouterGui("LsaGenerator");
    this.controlExec = control;
    controlExec.registerMessageProcessor(this);
    sourcePort = (short)controlExec.listenToRandomPort(this);
```

```
    destPort = (short)controlExec.SAAM_CONTROL_PORT;
    numberOfInterfaces = controlExec.getNumberOfInterfaces();
    timeBetweenLSAGenerations = LSA_PERIOD;
    interfaces = controlExec.getInterfaces();

    //add yourself as talker to ACE
    int channel=ProtocolStackEvent.FROM_LSAGENERATOR_TO_ACE;
    try{
      controlExec.addTalkerToChannel(this,channel);
    }catch(ChannelException ce){
      gui.sendText("Could not register as talker to ACE");
    }
    channel=ProtocolStackEvent.PACKETFACTORY_CHANNEL;
    try{
      controlExec.addTalkerToChannel(this,channel);
    }catch(ChannelException ce2){
      gui.sendText("Could not register as talker to PACKETFAC");
    }
    startInterfaceCheck();

  }//end constructor


  public LsaGenerator(ControlExecutive control,int generationTime){
    gui = new SAAMRouterGui("LsaGenerator");
    this.controlExec = control;
    controlExec.registerMessageProcessor(this);
    sourcePort = (short)controlExec.SAAM_CONTROL_PORT;
    destPort = (short)controlExec.SAAM_CONTROL_PORT;
    timeBetweenLSAGenerations = generationTime;
    interfaces = controlExec.getInterfaces();
    startInterfaceCheck();
  }//end constructor


  private void startInterfaceCheck(){
    Runnable check = new Runnable(){
      public void run(){
        while(true){
          try{
            synchronized(checkLock){
              while(!go){
                checkLock.wait();
              }
            }
            checkInterface();
          }catch(InterruptedException e){
            System.out.println(e);
            System.exit(1);
          }
        }//while
      }
    };
    intChecker = new Thread(check,"InterfaceChecker");
    intChecker.start();

    checkTimer = new Timer(2000,new ActionListener(){
```

```
        public void actionPerformed(ActionEvent event){
          gui.sendText("Timer expired. Interface check ...");
          synchronized(checkLock){
            go = true;
            checkLock.notify();
          }
        }
      });
    checkTimer.start();
  }


  private void checkInterface(){
    synchronized(interfaces){
      gui.sendText("Checking interfaces.");
      Enumeration enum=interfaces.elements();
      while(enum.hasMoreElements()){
        Interface temp=(Interface)enum.nextElement();
        //check the status of the interface
        if(!temp.getState()){
          //check if the interface lsa has been sent before if so dont
bother to
          //send it a second time
          if(!temp.isLSASent()){
            generateExtraLSA(temp);
            performLSACycle();
          }
        }//end if
      }//end while
      go=false;
    }//end synchronized block
  }//checkInterface


  public void setGenerationTime(int newTime){
    if(newTime < this.LSA_PERIOD){
      gui.sendText("Can not set generation time to a value smaller than
default");
      return;
    }
    timeBetweenLSAGenerations = newTime;
    timer.setDelay(timeBetweenLSAGenerations);
  }


  public int getGenerationTime(){
    return timeBetweenLSAGenerations;
  }


  public void startAction(){
    gui.sendText("Generating LSAs ");
    decideRouterID();
    timer = new Timer(timeBetweenLSAGenerations,this);
    timer.setInitialDelay(25000);
    timer.start();
    runner = new Thread(this,"Runner");
```

```
    runner.start();
  }

  public void actionPerformed(ActionEvent event){
    gui.sendText("Timer event occured");
    synchronized(myLock){
      started = true;
      myLock.notify();
    }
  }


  public String[] getMessageTypes(){
    return messageTypes;
  }


  public IPv6Address getRouterID(){
    return routerId;
  }


  public void run(){

    while(true){
      gui.sendText("Taking InterfaceLSAs...");
      performLSACycle();
      gui.sendText("LSAs are sent, I'm going to sleep");
      try{
        synchronized(myLock){
          gui.sendText("Waiting...");
          while(!started)
            myLock.wait();
        }

      }catch(InterruptedException ie){
          gui.sendText(ie.toString());
      }
    }//while(started)

  }//end method run()



  private synchronized void performLSACycle(){

    Vector iLsas=new Vector();

    Enumeration enum = interfaces.elements();
    while(enum.hasMoreElements()){
      Interface temp = (Interface)enum.nextElement();
      monitor = temp.getMonitor();

      InterfaceLSA lsa = monitor.getLSA();

      //if the status of the interface is down
      if(!temp.getState()){
```

```
        //check if the LSA is sent before
        if(!temp.isLSASent()){
          generateExtraLSA(temp);
          lsa.setLSAType(InterfaceLSA.REMOVE);
          checkIdChange(lsa.getIP());
          temp.setLSASent();
        }
      }

      iLsas.add(lsa);
    }//end while
    //construct the header of LinkStateAdvertisement for this router
    LSA = new LinkStateAdvertisement(routerId);
    LSA.insertInterfaceLSA(iLsas);

    if(idChange){
      decideRouterID();
      idChange = false;
    }

    //artik autocontrolexec e gonder
    //Now send the LSA via ControlExecutive
    //controlExec.sendLSA(LSA);
    int channel=ProtocolStackEvent.FROM_LSAGENERATOR_TO_ACE;
    MessageEvent me=new MessageEvent(
      this.toString(),
      this,
      channel,
      LSA);
    try{
      controlExec.talk(me);
      gui.sendText("LSA gonderildi "+"at "+System.currentTimeMillis());
    }catch(ChannelException che){
      gui.sendText("Could not send the LSA to ACE");
    }

    // if the this is server LSA should also be sent to itself
    if(controlExec.getIsServer()){
      long ts=System.currentTimeMillis();
      byte[] data=me.getMessage().getBytes();
      byte numbMes=1;
      data=Array.concat(numbMes,data);
      data=Array.concat(PrimitiveConversions.getBytes(ts),data);
      int channelP=ProtocolStackEvent.PACKETFACTORY_CHANNEL;
      ProtocolStackEvent pe=new ProtocolStackEvent(
        this.toString(),
        this,
        channelP,
        data);
      try{
        controlExec.talk(pe);
      }catch(ChannelException ce){
        System.out.println(ce);
      }
    }
    started = false;
  }//end method performLSACycle()
```

```
  public Message query(Message message){
    return message;
  }


  public void processMessage(Message message){
    gui.sendText("Received Message: "+message.toString());
    //here the server has requested that an LSA be sent.
    //Based on the parameters contained in the LsaRequest,
    //we would construct an LinkStateAdvertisement and
    //send it to the server
    //LinkStateAdvertisement lsa =
      //new LinkStateAdvertisement([parameters]);
    //sendLSA(lsa);
  }

  public void receiveEvent(SaamEvent se){
    ProtocolStackEvent event = (ProtocolStackEvent) se;
    int channel = event.getChannel_ID();
    gui.sendText("Received event from channel"+channel);
    //interfaceLSA came from the Interfacewe will add tje Interface LSA
to
    //LinkStateAdvertisement

  }//receiveEvent()



  private void decideRouterID(){
    //first, take the first Interface ip as the router id
    Enumeration enum1 = interfaces.elements();
    Interface temp;
    while(enum1.hasMoreElements()){
      temp = (Interface)enum1.nextElement();
      if(temp.getState()){
        routerId = temp.getID().getIPv6();
        break;
      }
    }//first while

    //now check if there is another ip that is bigger than the first one
    Enumeration enum2=interfaces.elements();
    while(enum2.hasMoreElements()){
      temp = (Interface)enum2.nextElement();
      if(temp.getState() & !routerId.equals(temp.getID().getIPv6())){
        byte[] tempBytes = temp.getID().getIPv6().getAddress();
        byte[] idBytes= routerId.getAddress();

        for(int ix=0;ix<IPv6Address.length;ix++){
          if(idBytes[ix]>tempBytes[ix]){
            break;
          }
          if(idBytes[ix]<tempBytes[ix]){
```

```
            routerId = temp.getID().getIPv6();
            break;
          }
        }//for
      }//if
    }//end 2nd while
  }//end method decideRouterID()


  /**
   *
   *
   */
  public void generateExtraLSA(Interface parent){

    LinkStateMonitor tempMonitor=parent.getMonitor();
    InterfaceLSA tempLSA=tempMonitor.getLSA();
    tempLSA.setLSAType(InterfaceLSA.REMOVE);
    LinkStateAdvertisement LSA=new LinkStateAdvertisement(routerId);
    LSA.insertInterfaceLSA(tempLSA);

    //I need to send other Interface LSAs since Server will look for
router id
    //change when it receives the LSA
    Enumeration enum=interfaces.elements();
    while(enum.hasMoreElements()){
      Interface iFace=(Interface)enum.nextElement();
      if(iFace.equals(parent)){
        continue;
      }
      LinkStateMonitor iMonitor=iFace.getMonitor();
      InterfaceLSA iLSA=iMonitor.getLSA();
      LSA.insertInterfaceLSA(iLSA);
    }//end while

    controlExec.sendLSA(LSA);

  }//end generateExtraLSA()


  private void checkIdChange(IPv6Address ip){
    if(routerId.equals(ip)){
      idChange=true;
      //decideRouterID();
    }
  }

  public String toString(){
    return ("LsaGenerator");
  }

}//end class LsaGenerator
```

APPENDIX D. LINKSTATEADVERTISEMENT CLASS SOURCE CODE

```
package saam.message;

import java.net.UnknownHostException;
import java.util.Vector;

import saam.net.*;
import saam.util.*;

/**
 * LinkStateAdvertisement are sent by the routers in a SAAM network to
 * keep the server abreast of their state.  The server makes routing
decisions
 * and adjustments to the network based on the LinkStateAdvertisements
it
 * receives.
 */
public class LinkStateAdvertisement extends Message{

  public static final int indexOfNumberOfInterfaces =17;

  private IPv6Address myIPv6 = new IPv6Address();

  private byte [] payload;

  Vector interfaceLSAs;

  byte[] data;
  byte numOfInterfaces =0;

  /**
   * Used by the Server.
   */
  public LinkStateAdvertisement(IPv6Address ip){
    super(Message.LSA);
    myIPv6=ip;
    payload = Array.concat(Message.LSA,payload);
    payload = Array.concat(payload,myIPv6.getAddress());
    payload = Array.concat(payload,numOfInterfaces);

    interfaceLSAs = new Vector();
  }


  public LinkStateAdvertisement(byte[] lsaByte){

    interfaceLSAs = new Vector();

    payload = lsaByte;
    int index = 0;
    //first byte is message type
    byte mType=payload[index++];
    //second item is ip address 16 bytes
    try{
```

```
      myIPv6 = new
IPv6Address(Array.getSubArray(payload,index,index+IPv6Address.length));
    }catch(UnknownHostException e){
      System.out.println(e);
    }
    index+=IPv6Address.length;
    //third item is Number of interfaces 1 byte
    numOfInterfaces = payload[index++];

    //now we will create "numOfInterfaces" InterfaceLSAs
    byte slpNum=0;
    int lengthOfLSA=0;

    for(int i=0;i<numOfInterfaces;i++){
      slpNum = payload[index+InterfaceLSA.SLPNumIndexOffset];
      lengthOfLSA = InterfaceLSA.headerLength + (slpNum*SLPLSA.length);
      InterfaceLSA lsa = new InterfaceLSA(
          Array.getSubArray(payload,index,index+lengthOfLSA));
      index+=lengthOfLSA;
      interfaceLSAs.add(lsa);
    }

  }


  public byte[] getBytes(){
    return payload;
  }


  /**
   * Returns the IPv6Address of the sender.
   * @return The IPv6Address of the sender.
   */
  public IPv6Address getMyIPv6(){
    return myIPv6;
  }

  /**
   * Returns the length of this Message.
   * @return The length of this Message.
   */
  public short length(){
    return (short)payload.length;
  }

  /**
   * Returns a <code>String</code> representation of this Message.
   * @return The <code>String</code> representation of this Message
   */
  public String toString(){
    return "Link State Advertisement";
  }


  public void insertInterfaceLSA(InterfaceLSA lsa){
```

```
    //increment the number of interface lsas
    numOfInterfaces++;

    //update the byte[] at number of interfaces
    payload[indexOfNumberOfInterfaces] = numOfInterfaces;

    payload = Array.concat(payload,lsa.getBytes());
    interfaceLSAs.add(lsa);
  }

  public void insertInterfaceLSA(Vector V){
    numOfInterfaces += V.size();

    payload[indexOfNumberOfInterfaces] = numOfInterfaces;

    for(int i=0;i<V.size();i++){
      InterfaceLSA temp = (InterfaceLSA)V.elementAt(i);
      payload = Array.concat(payload,temp.getBytes());
      interfaceLSAs.add(temp);
    }
  }

  public Vector getLSAs(){
    return interfaceLSAs;
  }

  public byte getNumberOfInterfaces(){
    return numOfInterfaces;
  }


}//end link state Advertisement
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX E. INTERFACELSA CLASS SOURCE CODE

```
package saam.message;

import java.util.Vector;
import java.util.Enumeration;
import java.net.UnknownHostException;
import java.lang.IndexOutOfBoundsException;

import saam.net.IPv6Address;
import saam.util.*;


public class InterfaceLSA {

  public static final byte UPDATE = 0;
  public static final byte ADD    = 1;
  public static final byte REMOVE = 2;


  public static final int SLPNumIndexOffset = 21;

  /**
   * 0 update this interface
   * 1 add this interface
   * 2 remove this interface
   */
  byte LSAtype;

  IPv6Address interfaceIP;

  //Kbps - 10000 is 10 MBps
  int bandwith;
  byte numOfSLPs =0; //default none

  Vector SLPs = new Vector();

  /**
   * IP length + bandwithlength(4) + lsatype (1) + numof SLPLSA (1)
   */
  public static final int headerLength = IPv6Address.length+ 4 + 1 + 1;

  byte [] bytes;

  /**
   * constructor of the class
   * if no type is specified this means it is an update as default
   */
  public InterfaceLSA(IPv6Address ipNum,int bandwith) {
    interfaceIP = ipNum;
    this.bandwith = bandwith;
    LSAtype = UPDATE;  //this is default

    bytes = Array.concat(LSAtype,bytes);
    bytes = Array.concat(bytes,ipNum.getAddress());
    bytes = Array.concat(bytes,PrimitiveConversions.getBytes(bandwith));
```

```
  }

  /**
   * constructs an InterfaceLSA with the specified InterfaceLSA type
   * @param ipNum IPv6Address of the interface
   * @param bandwith int bandwith of the Interface
   * @param type byte LSAtype
   */
  public InterfaceLSA(IPv6Address ipNum,int bandwith,byte type){
    interfaceIP = ipNum;
    this.bandwith = bandwith;
    LSAtype = type;

    bytes = Array.concat(LSAtype,bytes);
    bytes = Array.concat(bytes,interfaceIP.getAddress());
    bytes = Array.concat(bytes,PrimitiveConversions.getBytes(bandwith));
  }//end constructor


  public InterfaceLSA(byte[] data) throws IndexOutOfBoundsException{
    bytes = data;
    int index=0;
    LSAtype = bytes[index++];
    try{
      interfaceIP = new IPv6Address(Array.

getSubArray(bytes,index,index+IPv6Address.length));
    }catch(UnknownHostException uhe){
      System.err.println("Error in creating the IPv6Address for
InterfaceLSA");
    }
    index+=IPv6Address.length;
    bandwith =
PrimitiveConversions.getInt(Array.getSubArray(bytes,index,index+4));
    index+=4;
    numOfSLPs = bytes[index++];

    for(int i=0;i<numOfSLPs;i++){

      SLPLSA lsa = new
SLPLSA(Array.getSubArray(bytes,index,index+SLPLSA.length));
      index+=SLPLSA.length;
      SLPs.add(lsa);
    }

  }


  public void insertSLP(Vector slpVector){
    SLPs = slpVector;
    numOfSLPs += (byte)SLPs.size();
    byte[] slpByte=null;

    Enumeration enum = SLPs.elements();
    while(enum.hasMoreElements()){
      SLPLSA lsa = (SLPLSA)enum.nextElement();
      slpByte = Array.concat(slpByte,lsa.getBytes());
```

```
    }
    bytes = Array.concat(bytes,numOfSLPs);
    bytes = Array.concat(bytes,slpByte);
  }


  public void insertSLP(SLPLSA lsa){
    //increase the number of SLPs by 1
    numOfSLPs++;

    //update the number of SLPs
    bytes[SLPNumIndexOffset]=numOfSLPs;

    bytes=Array.concat(bytes,lsa.getBytes());

  }

  public Vector getSLPs(){
    return SLPs;
  }

  public byte[] getBytes(){
    return bytes;
  }

  public byte getNumOfSLPs(){
    return numOfSLPs;
  }

  public IPv6Address getIP(){
    return interfaceIP;
  }

  public void setLSAType(byte type){
    LSAtype = type;
    bytes[0]=type;
  }

  public byte getLSAType(){
    return LSAtype;
  }

  public int getBandwith(){
    return bandwith;
  }

  public String toString(){
    return "InterfaceLSA";
  }

}//end class InterfaceLSA

//end file InterfaceLSA.java
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. SLPLSA CLASS SOURCE CODE

```
package saam.message;

import saam.util.Array;
import saam.util.PrimitiveConversions;


public class SLPLSA {

  private byte SLPNum;
  private byte utilization;
  private short delay;
  private short lossRate;

  public static final double UTIL_UNIT = 0.5d;
  public static final double LOOSRATE_UNIT = 0.01d;

  //length of the SLPLSA
  public static final int length = 6;

  byte [] bytes;

  /**
   * constructor of the SLPLSA
   * @param number byte  number of  this SLP
   * @param util   byte  utilization of the SLP - between 0 and 200
   *                     200 means 100 percent utilization - unit
increment is 0.5
   * @param delay  short delay in this SLP in ms
   * @param loss   short loss rate of this SLP - between 0 and 10000
   *                     10000 means 100 percent utilization - unit
increment is 0.01
   */
  public SLPLSA(byte number,byte util, short delay,short loss) {

    SLPNum = number;
    utilization = util;
    this.delay = delay;
    lossRate = loss;

    bytes = Array.concat(SLPNum,bytes);
    bytes = Array.concat(bytes,utilization);
    bytes = Array.concat(bytes,PrimitiveConversions.getBytes(delay));
    bytes = Array.concat(bytes,PrimitiveConversions.getBytes(loss));
  }//end constructor

  /**
   * constructs a service level pipe link state advertisement
   * @param data byte[] that will be used to form a SLPLSA
   */
  public SLPLSA(byte[] data){
    if(this.length!=data.length){
      System.out.println("Error : wrong number of bytes");
    }
    bytes = data;    int index=0;
```

```
    SLPNum = bytes[index++];
    utilization = bytes[index++];
    delay = PrimitiveConversions.getShort(Array.
                getSubArray(bytes,index,index+2));
    index += 2;
    lossRate = PrimitiveConversions.getShort(Array.
                getSubArray(bytes,index,index+2));
  }//end constructor

  /**
   * returns the utilization of the slp
   * @return byte
   */
  public byte getUtilization(){
    return utilization;
  }//end method getUtilization()

  /**
   * returns the delay of the slp
   * @return short
   */
  public short getDelay(){
    return delay;
  }//end method getDelay()

  /**
   * returns the loos rate of the slp link state advertisement
   * @return short
   */
  public short getLossRate(){
    return lossRate;
  }//end method getLossRate()

  /**
   * returns the service level pipe number
   * @retun byte
   */
  public byte getSLPNum(){
    return SLPNum;
  }//end method getSLPNum()

  /**
   * returns the byte array representation of the data members
   * @return byte[]
   */
  public byte[] getBytes(){
    return bytes;
  }//end method getBytes()

  /**
   * return a string representing the class
   * @return String
   */
  public String toString(){
    return "\nService Level Pipe Link State Advertisement\n";
  }//end method toString()
```

```
}//end class
//end file SLPLSA.java
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G. INTERFACEFAILURE CLASS SOURCE CODE

```
package saam.message;

import java.net.UnknownHostException;

import saam.net.*;
import saam.util.*;


public class InterfaceFailure extends Message {

  public static final int length = IPv6Address.length+1;
  private IPv6Address interfaceIP;
  private byte[] bytes;

  public InterfaceFailure(IPv6Address ip) {
    super(Message.FAILURE);
    interfaceIP = ip;
    bytes = Array.concat(Message.FAILURE,bytes);
    bytes = Array.concat(bytes,interfaceIP.getAddress());
  }

  public InterfaceFailure(byte[] packet){
    super(packet[0]);
    int index=1;
    try{
      interfaceIP = new
IPv6Address(Array.getSubArray(packet,index,index+IPv6Address.length));
    }catch(UnknownHostException ex){
      System.out.println("Can not initialize IPv6address in
InterfaceFailure"+ex);
    }
  }

  public byte[] getBytes(){
    return bytes;
  }

  public short length(){
    return (short)bytes.length;
  }

  public IPv6Address getIP(){
    return interfaceIP;
  }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX H. SOURCE CODE OF LSA PROCESSING METHODS ADDED TO THE SERVER

```
/**
 * Receives link state advertisement messages from router and processes the
 * service level pipe status information that they contain. It begins by
 * checking to see if a router with the interface address described by this
 * LSA is known to the PIB. If such a router is known to exist, it then
 * checks to see if the service level pipe described by this LSA is known to
 * the PIB. If the service level pipe is known, then update its status.
 * Otherwise, add the SLP with the specified QoS characteristics. Finally,
 * update the effective QoS for the paths that pass over this service level
 * pipe by calling the determineEffectiveQoSForPaths().
 * @param router A representation of a router as defined by an LSA.
 */
public void processLSA(LinkStateAdvertisement LSA) {

 System.out.println("Started ProcessLSA");
 //who is the generating router
 IPv6Address routerId = LSA.getMyIPv6();
 gui.sendText("An LSA arrived at "+System.currentTimeMillis()+" from "+routerId.toString());
 long startTs,endTs;

 Vector IntLSAs = LSA.getLSAs();
 System.out.println("took the vector of interface lsa.");
 Vector ips = new Vector();


 boolean newRouter = false;
 int nodeId;
 String keyStr = routerId.toString();
 boolean check=IPv6ToIntIdTable.containsKey(keyStr);
 if(check){
  System.out.println("table contains the keystring");
 }
 //check if IPv6ToIntIdTable contains this router
 if(!check){
  System.out.println("This router is not in my table. \nTaking InterfaceLSAs from LSA message.");
  Enumeration enum = IntLSAs.elements();
  while(enum.hasMoreElements()){
   System.out.println("inside while.");
   InterfaceLSA tempLsa=(InterfaceLSA)enum.nextElement();
   IPv6Address tempIp=tempLsa.getIP();
   ips.add(tempIp);
  }
  //check if there is a router with these interafces
  //this means we removed an interface which was a router id earlier and
  //routerid has changed in the table
  nodeId=PIB.doesRouterExist(ips);
  //if the router is not in the pib it is a new one
```

```
  if(nodeId==this.ROUTERNOTINPIB){
    System.out.println("Router is a new router.");
    newRouter = true;
    nodeId=PIB.getNewNodeId();
    this.IPv6ToIntIdTable.put(keyStr,new Integer(nodeId));
  }
  else{ //it is not a new one this lsa is a second copy of the removal LSA
    System.out.println("LSA exit earlier");
    return;
  }
}

System.out.println("in ProcessLSA 1");

if(newRouter){
  //addd all interfaces to the PIB and compute the Paths
  int IdInt = ((Integer)IPv6ToIntIdTable.get(keyStr)).intValue();
  InterfaceLSA newInterface=null;
  Enumeration enum=IntLSAs.elements();
  while(enum.hasMoreElements()){
    newInterface=(InterfaceLSA)enum.nextElement();
    this.checkAndAdd(IdInt,newInterface);
  }
  findAllPossiblePaths();
  determineEffectiveQoSForPaths();
  return;
}

System.out.println("in ProcessLSA 2");

//it may be a new or an old router take the int id of the router
nodeId = ((Integer)IPv6ToIntIdTable.get(keyStr)).intValue();

Enumeration lsaInterfaceEnum = IntLSAs.elements();
while(lsaInterfaceEnum.hasMoreElements()){
  InterfaceLSA curInterface=(InterfaceLSA)lsaInterfaceEnum.nextElement();
  byte type = curInterface.getLSAType();
  gui.sendText("Type of the InterfaceLSA is "+(int)type);
  switch(type){
    case InterfaceLSA.ADD:
      checkRouterId(routerId,IntLSAs);
      checkAndAdd(nodeId,curInterface);
      if(!newRouter){ //another interface is added to the router
                //PIB will be updated
        findAllPossiblePaths();
        determineEffectiveQoSForPaths();
      }
    break;

    case InterfaceLSA.UPDATE:
      updatePIB(nodeId,curInterface);
    break;
```

```
    case InterfaceLSA.REMOVE:
      checkRouterId(routerId,IntLSAs);
      removeInterfaceFromPIB(nodeId,curInterface);
    break;

    default:
      gui.sendText("Interface LSA type is not a recognized type.");
   }//end switch
  }//end while

  System.out.println("end of ProcessLSA");
 } //end processLSA



private void checkRouterId(IPv6Address routerId,Vector iLsaVector){
  IPv6Address tempIP;
  IPv6Address tempId=new IPv6Address();
  InterfaceLSA tempIntLsa;
  byte[] idBytes;
  byte[] tempBytes;
  Enumeration enum=iLsaVector.elements();
  while(enum.hasMoreElements()){
    tempIntLsa = (InterfaceLSA)enum.nextElement();
    if(tempIntLsa.getLSAType()==InterfaceLSA.REMOVE){
      continue;
    }
    tempIP=tempIntLsa.getIP();
    idBytes=tempId.getAddress();
    tempBytes= tempIP.getAddress();
    for(int i=0;i<IPv6Address.length;i++){
      if(idBytes[i]>tempBytes[i]){
        break;
      }
      if(idBytes[i]<tempBytes[i]){
        tempId=tempIP;
        break;
      }
    }
  }

  if(tempId.equals(routerId)){//there is no change in id
    return;
  }
  //there is change in the router id
  //old router id has to be changed from the table
  int knownId=((Integer)IPv6ToIntIdTable.get(routerId.toString())).intValue();
  IPv6ToIntIdTable.remove(routerId.toString());
  routerId=tempId;
  IPv6ToIntIdTable.put(tempId.toString(),new Integer(knownId));
}//end checkRouterId()
```

```
private void checkAndAdd(int nodeId,InterfaceLSA curInterface){

  IPv6Address ip=curInterface.getIP();
  int bandwidth=0;
  //Is this interface in my Path Information Base
  if(!PIB.doesInterfaceExist(ip)){
    bandwidth = curInterface.getBandwith();

    if(!PIB.doesLinkExist(ip)){
      PIB.addLink(ip,bandwidth);
    }//end inner if

    //now add interface
    PIB.addInterface(nodeId,ip);

    //add service level pipes
    byte slps=curInterface.getNumOfSLPs();
    Vector slpVector=curInterface.getSLPs();
    for(int i=0;i<slps;i++){
      gui.sendText("Adding SLP");
      SLPLSA slpLsa = (SLPLSA)slpVector.elementAt(i);
      byte slpNumber = slpLsa.getSLPNum();
      byte utilization = slpLsa.getUtilization();
      short delay = slpLsa.getDelay();
      short lossRate = slpLsa.getLossRate();
      PIB.addSLP(ip,slpNumber,INITIALDELAY,INITIALLOSSRATE,INITIALTHROUGHPUT);
      PIB.updateSLP(ip,slpNumber,delay,(int)(lossRate/100),(utilization/2));
    }
  }//end if
}

private void updatePIB(int nodeId,InterfaceLSA iLsa){
  byte slps=iLsa.getNumOfSLPs();
  Vector slpVector=iLsa.getSLPs();
  IPv6Address ip=iLsa.getIP();
  for(int i=0;i<slps;i++){
    SLPLSA slpLsa=(SLPLSA)slpVector.elementAt(i);
    byte slpNumber=slpLsa.getSLPNum();
    byte utilization=slpLsa.getUtilization();
    short delay = slpLsa.getDelay();
    short lossRate = slpLsa.getLossRate();
    PIB.updateSLP(ip,slpNumber,delay,(int)(lossRate/100),(utilization/2));
  }
}


private void removeInterfaceFromPIB(int nodeId,InterfaceLSA curInterface){
  gui.sendText("Removing interface from PIB.");
  removePathsTraversingInterface(curInterface);
```

```
    removeLinkFromPIB(curInterface.getIP());
    removeInterfaceFromNode(curInterface.getIP());
  }

  private void removePathsTraversingInterface(InterfaceLSA iLsa){
    gui.sendText("Removing Paths using the interface from PIB.");
    for(int i=0;i<iLsa.getNumOfSLPs();i++){
      Vector pathIds=PIB.getAllPathIdsThatTraverseSLP(iLsa.getIP(),i);

      ClassObjectStructure cos=(ClassObjectStructure)PIB;
      cos.deletePathsTraversingInterface(pathIds);

    }
  }

  private void removeLinkFromPIB(IPv6Address ip){
    gui.sendText("Removing linkof the interface from PIB.");
    IPv6Address netIP=ip.getNetworkAddress();
    ClassObjectStructure cos=(ClassObjectStructure)PIB;
    cos.links.remove(netIP.toString());
  }

  private void removeInterfaceFromNode(IPv6Address ip){
    gui.sendText("Removing Interface from nodes.");
    ClassObjectStructure cos=(ClassObjectStructure)PIB;
    cos.nodes.remove(ip.toString());
  }
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Keshav, S., *An Engineering Approach to Computer Networking*, pp.290-312, Addison-Wesley, 1998.

[2] Peterson, Larry L. and Davie, Bruce S., *Computer Networks, A Systems Approach*, Morgan Kaufmann, 2000.

[3] Awerbuch, Baruch, Du, Yi, and Shavitt, Y., "The Effect of Network Hierarchy Structure on Performance of ATM PNNI Hierarchical Routing," proceedings of the Seventh International Conference on Computer Communications and Networks, IEEE, 1998, pp.73-78.

[4] Song, Yi, Cypher, D., and Su, D., "Simulation and Performance of PNNI ATM Networks," proceedings of the Seventh International Conference on Telecommunications Systems Modeling and Analysis, 1999, ppp.387-401.

[5] Xie, Geoffrey G., Hensgen, Debra, Kidd, Taylor, Yarger, J., "A Study of the Feasibility of Maintaining a Comprehensive Path Information Base" 14 May 1998. [http://www.cs.nps.navy.mil/people/faculty/xie/pub]

[6] Vrable, Dean, Yarger, John, "The SAAM Architecture: Enabling Integrated Services", Thesis September 1999, Computer Science Department Naval Postgraduate School

[7] Xie, Geoffrey G., Hensgen, Debra, Kidd, Taylor, and Yarger, John, "Efficient Management of Integrated Services Using a Path Information Base," 14 May 1998.
[http://www.cs.nps.navy.mil/people/faculty/xie/pub].

[8] Xie, Geoffrey G., Hensgen, Debra, Kidd, Taylor, and Yarger, John, "SAAM: An Integrated Network Architecture for Integrated Services," paper presented at the 6th IEEE/IFIP International Workshop on Quality of Service, Napa, CA, May 1998.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Chairman,Code CS.....1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

4. Dr. Geoffrey Xie.....1
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5100

5. Dr. Bert Lundy 1
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5100

6. Mr. Cary Colwell.....1
   Computer Science Department, Code CS
   Naval Postgraduate School
   Monterey, California 93943-5100

7. H. Huseyin Uysal.....2
   Armagan Mah.
   Sevilay Sok.
   Yamut Apt. D.5
   Meram/Konya
   Turkey

32 473NPG
TH 579
11/02 22527-200 NLE

DUDLEY KNOX LIBRARY
3 2768 00404081 6